1995

A security extension to the TCP/IP protocol under Linux

David T. Denton
University of Wollongong

A Security Extension to the TCP/IP Protocol under Linux

A thesis submitted in fulfilment of the

requirements for the award of the degree of

Honours Master of Science

(Computer Science)

from

THE UNIVERSITY OF WOLLONGONG

by

David T. Denton, B.Sc. (UNSW)

Department of Computing Science

1995

I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and I am not aware of any similar work either prior to this thesis or currently being pursued.

D. Denton

Abstract

An enhancement to TCP (Transmission Control Protocol) is proposed to give additional security between cooperating client/server programs.

Several modes of operation are available including: program controlled mode where new ioctl commands are used to provide explicit control over the actions taken by the kernel in the provision of security services, client or server controlled mode where one of the client/server pair initiates and controls security features without the knowledge or intervention of the other party and finally kernel controlled mode where two kernels will establish and maintain secure communication without intervention of either client or server program.

To achieve client or server controlled and kernel controlled operation, three new TCP option fields have been defined to allow the passage of security setup information at session establishment time. The threeway handshake session setup of TCP is used as a vehicle to piggyback information used to establish the type of encryption scheme in use as well as encrypted session keys.

Flexibility in the type of encryption schemes used is permitted with open access to the methods of defining new schemes to be used and mapping of these schemes to new scheme numbers within the TCP option fields. Schemes implemented as part of this work include an optimised version of DES (Data Encryption Standard) in both codebook and feedback mode, Triple DES in both codebook and feedback modes, LOKI is used in codebook mode and a new stream oriented cipher called Sapphire is used to compare purpose built stream ciphers with more traditional block methods.

The implementation of the protocol is described as it applies to the Linux operating system kernel structures, program code and the new formats for TCP. Some performance issues are then canvassed as they apply to the various schemes employed. The performance measures apply to client/server programs exchanging data either on a single host or between adjacent hosts on an otherwise unloaded ethernet network. When other than trivial encryption schemes are employed the cost of encryption becomes dominant and network bandwidth restriction are not an issue.

The procedures required to add a new encryption scheme using the formats and entry points of a canonical scheme is then described. Using this description it is possible to implement any new scheme that can be expressed in this canonical format.

Possible suggestions for enhancements to overcome potential bottlenecks are then discussed and a conclusion is given.

TABLE OF CONTENTS

1. Introduction

With the advent of allpervasive global network technology the requirement for improved security over existing networks will increase. Authorities are adopting principles of privacy and confidentiality over information held by others. Increasingly the onus of responsibility for demonstrating the appropriate need for, handling of, and disposal of such information is falling on the holders of such information.

The handling of information includes (but is not limited to) gathering and dissemination electronically. It is possible that the gathering and dissemination of such information may require transmission over various types of networks. In general, if no special action is taken, information is transmitted and received in a format that can be readily interpreted by a determined third party. Consider the situation where a sensitive database is contained on a host in a format that is encrypted on disk. In a static sense the data is protected against unauthorised disclosure but in a dynamic sense when remote access is required, the DBMS system would, for example, decrypt data on the DBMS host and present it to users in a format that could be handled by existing network and application protocols such as TCP, telnet and rlogin. The transmission of such data over shared network facilities may pose an unacceptable risk of disclosure, particularly if such facilities are shared between organisations with different statutory requirements and cultures. The need to provide an end-to-end secure service has been discussed by Brown [12] and Ioannidis [18] and is discussed below.

This work takes a similar approach at a different level and will enhance an existing transport protocol to enable users to pass data between applications with an increased level of security and hence address some of the possible concerns of disclosure and unauthorised access discussed above. As a starting point for the work, it is assumed that we have access to a notionally secure application and host operating system environment. Should either of these be compromised then any level of network security will be to little avail.

Several concepts have been introduced and will now be expanded on. These include the notion of security and the principleles of encryption that can be used to enhance security. Next, concepts concerning networking are discussed together with a more indepth consideration of the Transmission Control Protocol. Previous work in the general area of network security is then discussed along with some of the pros and cons of such approaches.

Any networking software requires an underlying operating system and hardware to run on. In this instance the Linux operating system was chosen running on a typical Intel® 80486 PC. A general description of Linux is provided together with a brief tour of how a simple network application would proceed during execution in the kernel. A description of the enhanced functionality proposed is then presented along with descriptions of the architectural and implementation changes that are required to make it possible. Some performance measures are provided to estimate the cost of such enhancements. A conclusion is provided together with some possible future extensions.

2. Security and Encryption

Security is linked with the idea of protection. An entity is considered valuable if it is worth spending effort protecting it. The act of protection is security. Security comes in many forms. Physical security is concerned with protection of real assets and can include such things as restricting access with fences, locked rooms, guards, sealed channels, fire suppression, tamper proof terminals etc. Logical security is concerned with the protection of information and is used to complement the protection provided by physical security. The science of logical security as it applies to data is known as cryptography.

Cryptography can provide a number of functions. These include the ability to hide information in messages providing privacy and the ability to identify the source of messages providing authentication. The fundamental tool used in cryptography is encryption. Encryption provides a method of transforming a message in such a way that it is readable by authorised receivers and unreadable by anyone else. The basic property of the encryption transformation is that it is not easily invertible without access to a parameter known as the enciphering key. From [34] we have a formal definition of a cryptosystem (an implementation of an encryption transformation) as a single parameter family of invertible functions,

$$\mathbf{E_K}; K \in \mathbf{K}$$

where **K** is the keyspace, which is of finite length. If **M** is the message space and **C** is the cryptogram space, the we must have the following properties:

- An enciphering algorithm

  $$E_K: \mathbf{M} \rightarrow \mathbf{C}$$

  for any fixed encryption key $K \in \mathbf{K}$ is an invertible transformation of the message space into the cryptogram space,

- There is an inverse algorithm $E_K^{-1} = D_K$ called the decryption algorithm.

  $$D_K: \mathbf{C} \rightarrow \mathbf{M}$$

  Such that $D_K(C) = D_K[E_K(M)] = M$;

- The keys should uniquely define the enciphered message,

$$E_{K_1}(M) \neq E_{K_2}(M) \quad \text{if} K_1 \neq K_2$$

Whereas cryptography is concerned with designing and implementing secure encryption algorithms, cryptanalysis is concerned with the recovery of messages without recourse to the cipherkey. The cryptanalyst may have multiple motives including the ability to read ciphered messages, modify messages without authority, pose as a legitimate user, replay messages or transactions or deny service. The main thrust of this paper is to thwart attempts to read messages without authority, however certain other aims mentioned above may also be less effective as a result of these enhancements.

2.1. Symmetric Encryption

Symmetric encryption is a scheme where two parties share a common cipherkey K and a known encryption and decryption scheme. The encryptor forms a cryptogram by passing a message M through the encryption scheme using the cipherkey K producing ciphertext C. The ciphertext is then transmitted to the receiver who applies the decryption scheme using the same cipherkey. This reveals the original message.

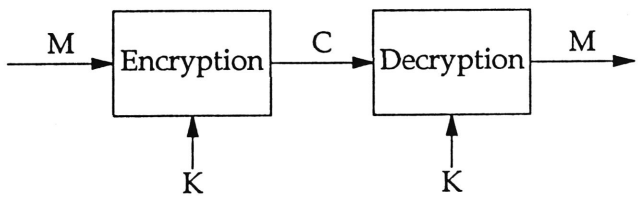


Fig #1 - Symmetric Encryption

2.1.1. DES

The Data Encryption Standard (DES) ANSI-X3.92-1981 provides an implementation of a symmetric encryption scheme. It uses a sixteen round system of permutations and

substitutions with a 56 bit key and 64 bit blocks of data. The permutations are provided by bit selection tables for bitwise reordering. The substitution function is provided by the so called S-boxes which take 6 bit input values that have previously been expanded from 4 bits and produces 4 bit output values. Each round uses the permutation and substitution function along with a permuted key choice based on the original key.

ANSI-X3.106-1983 Data Encryption Standard - Modes of Operation, provides two methods of using DES. In Electronic Codebook Mode (ECB) input is blocked in 64 bit blocks and each block is encrypted using the same 56 bit cipherkey to produce 64 bit output blocks.

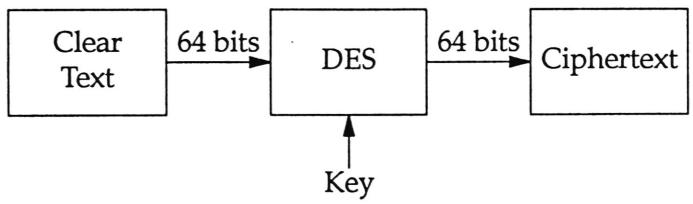


Fig #2 - DES ECB Mode

In Cipher Block Chaining mode (CBC) the results of previous encryptions are fed back to the current input. As in ECB input data is blocked into 64 bit blocks. The first block is combined with an initial vector and encrypted to give the first cryptogram. The cryptogram is forwarded to the receiver and also replaces the initial vector. The second block is now combined with the initial vector and encrypted giving the second cryptogram and replacing the initial vector and so on.

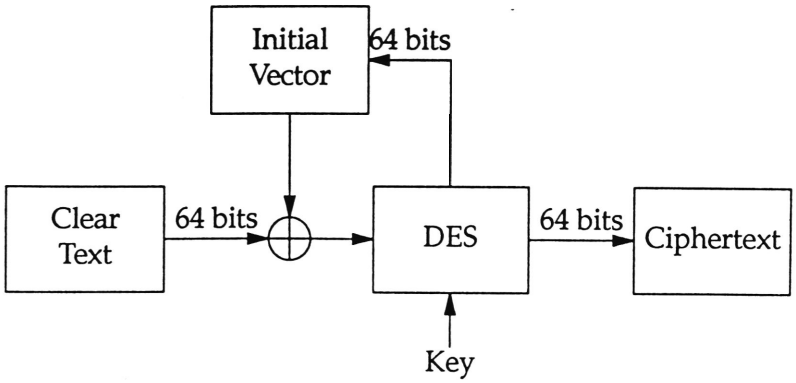


Fig #3 - DES CBC Mode

There has been considerable controversy over the perceived secrecy in the early development of DES [38]. There was concern that the details of the design principals of the S-boxes were kept secret and that the 56 bit key was too short. To overcome this second problem it is possible to use DES in what has become known as Triple-DES. In this mode two 56 bit keys and three phases of encryption are used. The first phase encrypts with the first key. The second phase decrypts with the second key. The third phase again encrypts with the first key. This mode of operation provides enhanced security compared to ordinary DES.

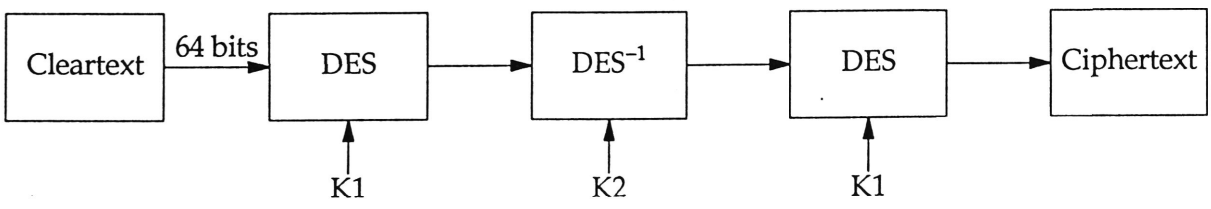


Fig #4 - Triple-DES

Other examples of symmetric algorithms include LOKI, FEAL and IDEA.

2.2. Asymmetric Encryption

Symmetric encryption assumes that the legitimate senders and receivers of a message have access to the secret key required for enciphering and deciphering. It is possible to arrange for an encryption scheme to be designed such that the key has two parts, one of these parts is public and globally known, the other part is private and known only to the receiver. A ciphertext message is constructed by encrypting with the public key and retrieved by decrypting with the private key. Such schemes are known as asymmetric encryption schemes.

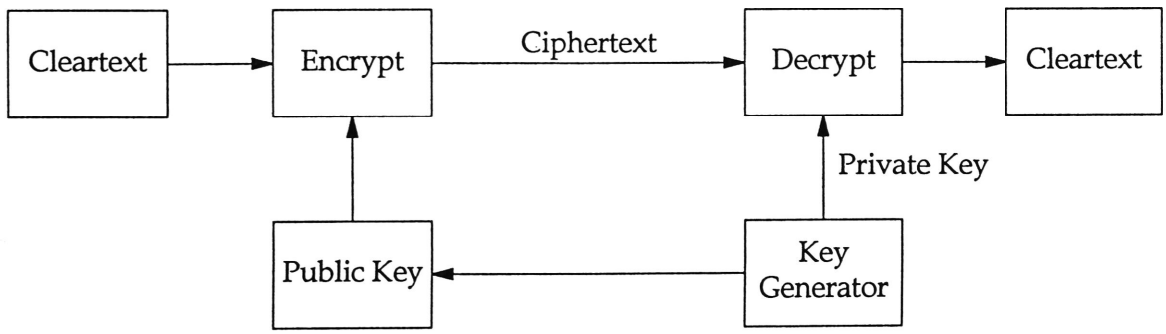


Fig #5 - Asymmetric Encryption

Typically, asymmetric schemes use combinations of numeric problems thought to be intractable (such as factorisation and discrete logarithm) to implement the encrypting and decrypting functions. An example of an asymmetric scheme is RSA which uses the fact that it can reveal enough information to compute certain functions over discrete rings but maintain the necessary factors to decode the ciphertext in secret.

The major disadvantage of asymmetric schemes designed to date is that to provide the required level of intractability the basic encryption and decryption processes tend to be computationally expensive compared to symmetric algorithms such as DES. For this reason asymmetric schemes tend to be used to distribute keys to be subsequently used by symmetric schemes.

2.3. Block Cipher vs Stream Cipher

The encryption schemes discussed so far have assumed that input data can be easily blocked in 64 bit blocks. This method of operation can be inconvenient in a networked environment where data may be transmitted on a byte (or even bit) oriented boundary. Adapting a block cipher scheme to a stream cipher scheme is relatively simple and involves using the block oriented cipher to provide a pseudo-random bit sequence that can be XOR'ed with any output stream to provide an encrypted stream.

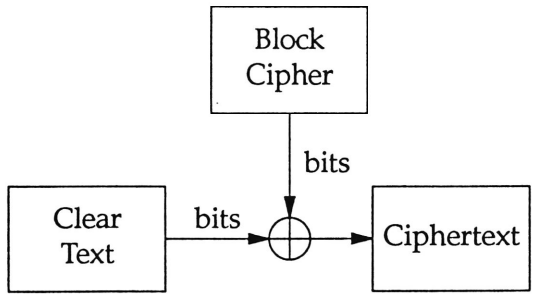


Fig #6 - Stream Cipher

Alternatively, special purpose byte (or bit) oriented stream ciphers have been designed that operate without the requirement to use a pseudo random number generator as a source of bits. Examples of such ciphers include RC4, a proprietary system from RSA Data Security Inc. and the Sapphire Stream Cipher by Johnson [22]. The Sapphire Stream Cipher can be conceptually compared to a game of cards. The deck has 256 cards with face values ranging from 0 to 255. The initial deck is shuffled with values based on the user's key, which can be up to 255 bytes long. In operation, five cards are drawn from the deck. The draw is based on three index variables (rotor , ratchet, avalanche) and the previous value of input, both plain and encrypted. These five cards are rotated in the deck and the three index values updated. Two of these updates (ratchet and avalanche) are based on values drawn and the third (rotor) is simply an increment. Now five cards are drawn based on the new values of the

index variables and the existing values of the last input, both plain and encrypted. The face values of the five drawn cards are added modulus 256 and used to select another card whose face value is used as an index into the deck once again. This final draw provides a suitable byte to XOR with the input stream. Decryption is identical to encryption with the exception that the resulting XOR on the enciphered text produces the original plain text.

The robustness of the Sapphire Stream Cipher is yet to be demonstrated. (Various follow ups to [22].)

3. Networking

The basic function of a network is to provide connectivity between two processes that wish to communicate. These processes may ultimately be human or machine and it is possible that there may be more than two processes. The International Standards Organisation (ISO) has provided a model known as the Open Systems Interconnect (OSI) that allows a layered approach to network connectivity to be developed.

3.1. The ISO Reference Model of Open Systems Interconnection (OSI)

This seven layered model has been established by the International Standards Organisation (ISO) according to several principals. These include:

- Each layer creates a new level of abstraction.
- Each layer performs a well defined function (or functions).
- Each layer should be able to define International Protocol Standards at its boundary.
- Information flow between layers should be minimal.

ISO Reference Model

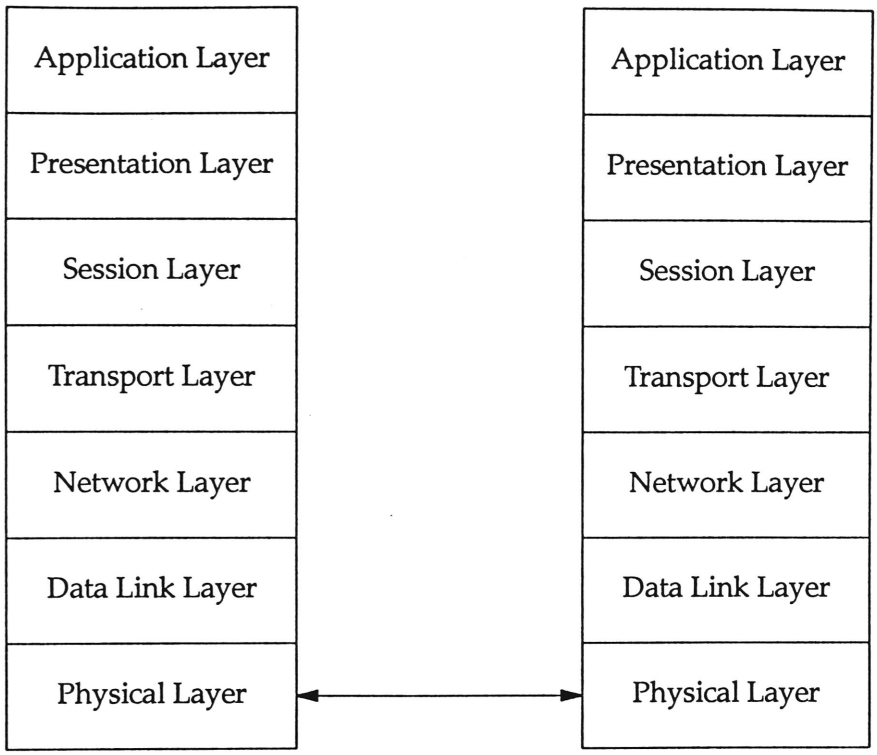


Fig #7

3.1.1. The Layers of OSI

The seven layers in the OSI Model are:

- The Physical Layer
- The Data Link Layer
- The Network Layer
- The Transport Layer
- The Session Layer
- The Presentation Layer
- The Application Layer

Generally they are shown as a stack of protocol blocks with the Application Layer at the top and the Physical Layer at the bottom.

Each layer is briefly described below:

3.1.1.1. The Physical Layer

The primary concern of the physical layer is the format and transmission of bits of data. Various methods of transmitting bits can be used, as can various media. Issues addressed within this layer include mechanical, procedural and electrical interfacing as well as capacity and timing. The **X.21** standard, which was approved in 1976, is an example of a Physical layer protocol which specifies how a DTE (Data Terminal Equipment) sets up and clears down calls with a DCE (Date Circuit-Terminating Equipment).

3.1.1.2. The Data Link Layer

This layer is concerned with providing a reliable and efficient method of communicating a set of bits, typically known as a frame, between two adjacent pieces of network hardware. In this context adjacent means that both pieces of hardware are joined by a conceptual "wire" and that bits entering one end of the "wire" will appear in the same order at the other end. The "wire" can be physical cable (metal or fibre optic), infrared, laser, satellite channel, etc as long as it appears "wire like".

While it appears that sending and receiving frames along a piece of wire is simple, protocols at this level must deal with errors such as: line noise (which scrambles bits within a frame or loses entire frames), the limited reception capacity of a receiving piece of network hardware (which can cause frames to be dropped or otherwise ignored), the underlying capacity of the "wire" and the possibility that frames may be duplicated by error (say, echo on a line) or design (a transmitter thought a frame was lost when it wasn't and resent it). Protocols which achieve the above goals are candidates for Data Link Protocols. Some examples include **HDLC** (High Level Data Link Control) and its subset **LAPB** (Balanced Link Access Procedure) which is often used with X.21 in achieving point to point or multipoint wide area

connection. The standards of the IEEE 802 Committee deal with Data Link Protocols for local area networks including Ethernet and Token Ring.

3.1.1.3. The Network Layer

The function of the Network Layer is to deliver, via one or more "hops" along Data Link Layers, a set of data bits known typically as a **Packet** or **Datagram** from a source "Host" to a destination "Host".

On receiving a data packet from the next highest level in the protocol stack (the Transport Layer) the network layer will choose the appropriate Data Link Layer interface and form a frame consisting of the data associated with the packet, the packet header (network addressing information that will not change as a result of hopping) and the Data Link header (interface addressing information for the current hop). This frame is then passed to the data Link Layer for transmission across the physical media. This process is continued on a hop by hop basis until the destination is reached at which time the packet is handed to the next highest layer of the protocol for processing.

The Network Layer is responsible for routing between source and destination hosts. It is also typically responsible for congestion control.

In the area of switched packet networks **X.25** represents an International Standard for the implementation of a Network Protocol. Within the TCP/IP Internet suite **IP** (Internet Protocol) can loosely be considered a Network Layer protocol.

3.1.1.4. The Transport Layer

The Transport Layer provides a reliable and efficient end to end message service. Whereas the Network Layer is concerned with the concept of a packet, whose characteristics are defined by underlying layers, the Transport Layer deals with messages which may be of arbitrary length and are defined by higher levels of the protocol. This requires the Transport Layer to provide multiplexing, fragmentation, de-fragmentation and buffering functions. While the Network Layer attempts to provide consistent delivery it can not guarantee such

services. The Transport Layer must therefore provide a service without error, loss, rearrangement or duplication of messages.

An International Standard for the Transport Layer is provided by **X.75**. In the TCP/IP Internet suite **UDP** can be loosely considered a Transport Protocol but lacks some of the reliability features described above. **TCP** includes such facilities as well as many others that could well be considered Session Layer features.

3.1.1.5. The Session Layer

The Session Layer builds on services provided by the Transport Layer. In particular the provision of user addressing. In general the Transport level provides host-to-host message delivery and the session layer provides user-to-user delivery by adding user identification to message headers thus allowing the receiving host to determine which user owns an incoming message. The Session Layer may also provide crash recovery, synchronisation, checkpointing facilities, transport layer congestion control or dynamic reconfiguration control over multiple transport paths.

Within the TCP/IP Internet suite **TCP** provides much of the Session Level functionality described above.

3.1.1.6. The Presentation Layer

The Presentation Layer can be used to provide a range of add-in functions that are not readily adaptable at lower levels of the protocol stack. These services are, by their nature, added as needed, usually through shared libraries or additional operating system services.

Traditionally, Encryption and Privacy measures are seen as belonging to the Presentation Level since it is desirable to encipher messages as close to their origin as possible.

Other services that can be provided at the Presentation Level include data compression, Virtual Terminal functionality, Virtual File Systems and character set translation.

In the Unix environment Virtual Terminal Protocols include such things as **Telnet** and **X11**, while Virtual File Systems are provided by **NFS** and **ftp**. Many systems offer "3270 type"

emulation packages that provide character translation function between ASCII and EBCDIC character sets.

3.1.1.7. The Application Layer

The Application Level can be used to provide services that have a specific functional requirements. This compares with the Presentation Level which provides general tools within the "Distributed Operating System" theme.

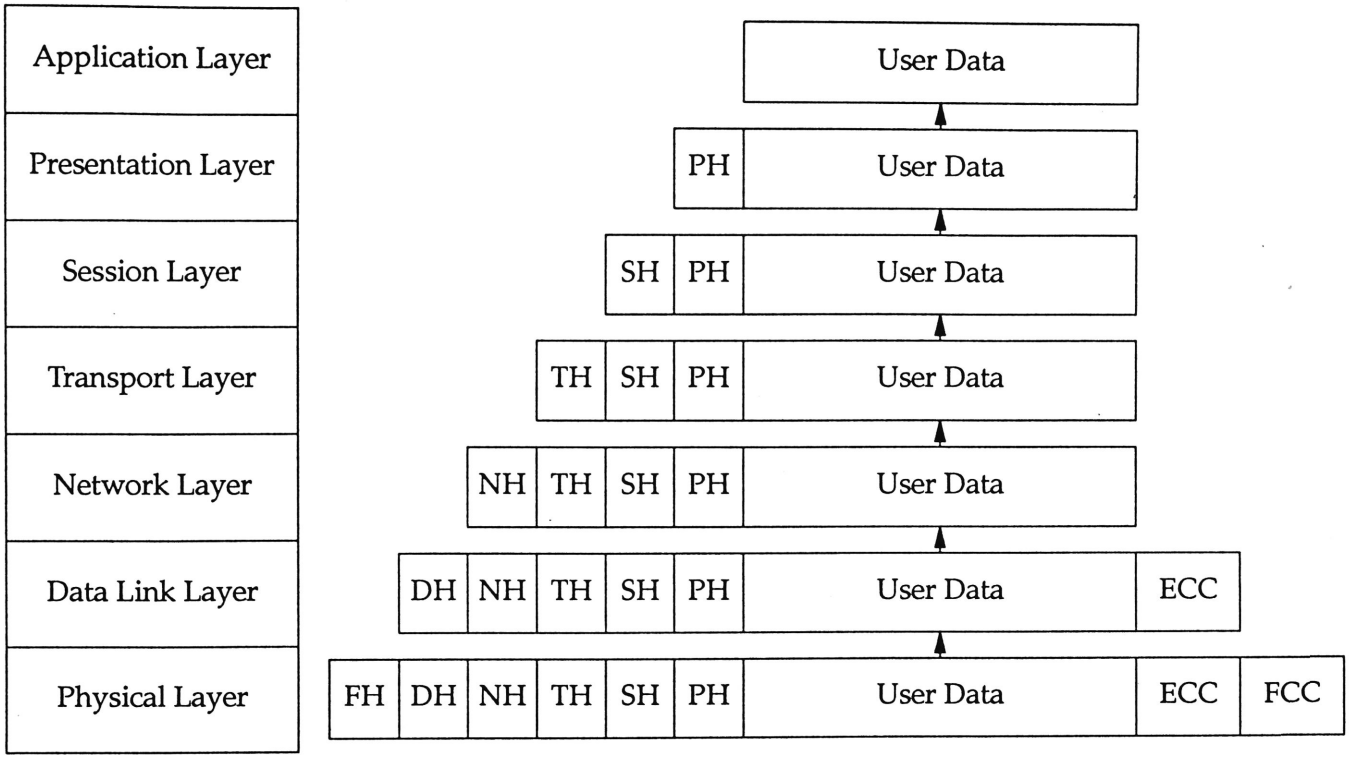
Services may build on those provided by the Presentation Level and may include things such as distributed database management and its associated transaction processing facility, mail services and protocols, directory services and protocols, time services, authentication services, hierarchical storage management, news services and hypertext services.

The Unix environment mail services are provides by **SMTP** [15] (among others), time services by **NTP** [10] , authentication services by **Kerberos** [36] , news services by **NNTP** [15].

In the International Standards area mail protocols include **X.400** with directory services provided by **X.500**.

3.1.2. Delivery of Data through the Protocol Stack

Typically each layer places a header on data that it receives from the next highest layer. When it reaches its final destination headers are removed one layer at a time and the final data is delivered to its "end-user". This is illustrated in the following diagram.



PH - Presentation Header, SH - Session Header, TH - Transport Header, NH - Network Header, DH - Data Link Header, FH - Physical Header, ECC - Error Correcting Code, FCC - Frame Check Code.

Fig #7 - Layer Headers

3.2. Security Issues

Each layer of the protocol stack faces its own set of problems and solutions in the area of network security. Some of these issues will be addressed in the following paragraphs with reference to examples of specific implementations at each level. (This is not meant to be an exhaustive coverage but rather an introduction).

3.2.1. Physical Layer Security Issues

The issues of concern at the Physical Layer of the protocol stack are in many ways similar to the physical security issues faced by system implementors of all computer systems. To be

secure from an attack that either collects intelligence or denies access, any system (computer or otherwise) must employ techniques of defence well known outside the computing industry. These include such things as: physically restricted areas, all round security in depth, segregated personnel responsibility, physical monitoring etc. In a network environment however, some of these techniques are difficult to implement and must be considered in the context of the geographically dispersed nature of the network. Two techniques that relate to the physical security of networks are presented below as examples.

3.2.1.1. TEMPEST

The acronym TEMPEST has two meanings. Firstly the offensive meaning used by intelligence gatherers, Transient Electromagnetic Pulse Surveillance Technology, which works by monitoring emissions of electromagnetic radiation (EMR) from devices and reconstructing useful information from these emanations. Secondly TEMPEST has a defensive meaning, Transient Electromagnetic Pulse Emanation Standard, which are techniques used to hide, distort or baffle EMR so that offensive TEMPEST techniques will fail.

All electromagnetic devices emit electromagnetic radiation. Some of this radiation is used to transmit information but much is simply radiated off into space and this can be intercepted and reconstructed into coherent form. For example, VDUs typically work by using an electron gun to scan across a screen many times per second refreshing pixels which then glow for a short time before the next scan. The firing of the electron gun causes emissions of EMR which can be detected and intercepted, with the intelligence gatherer simply synchronising their own electron gun with the targets', thus replicating the image of the target screen on their own screen. Since all devices have characteristic signatures it is possible to "tune in" to a particular device at a distance of up to one kilometer.

Although more difficult, the same techniques can by used for memory devices and I/O channels (at much shorter distances than VDUs) to reconstruct memory images and I/O traffic on target systems.

In its defensive mode TEMPEST attempts to mask EMR emissions so that offensive TEMPEST techniques will fail. These techniques can be applied at a number of levels. Devices themselves can be baffled to reduce the level of EMR they radiate and hence reduce the range at which offensive TEMPEST is effective. At the next level, rooms, buildings or entire installations can be baffled so that non TEMPEST rated devices can not emanate EMR beyond secured boundaries. This could be effective in the case of a local area network where individual shielding would be impractical.

While TEMPEST provides a significant level of physical security it is very expensive to implement and maintain and generally is only available to military and intelligence community users.

3.2.1.2. Phreaking

Just as government and company based networks need security, carrier based networks (telecoms) also need network security. This has been demonstrated by the emergence of a technique known as *phreaking*.

Phreaking works by exploiting known (and sometimes undocumented) signaling and diagnostic features of modern telephone exchanges to gain unauthorised access to carrier services. Generally an acceptable level of human speech can be carried between the 700 and 900 Hz range, by using frequencies well beyond these limits the national carriers can place routing, diagnostic and signaling information on the same physical circuits, thus avoiding duplication of infrastructure (data and control) on long distance networks. With a knowledge of such standards as the CCITT Signaling System 7 and the CCITT C5 C6 C7 which deal with the way PABXs and Local Exchanges communicate plus access to some relatively cheap hardware for generating the required frequency tones, it has been demonstrated that unauthorised access within a variety of countries is possible. This uncontrolled access to national carrier facilities represents a potential security threat as well as a loss in revenue.

3.2.2. Data Link Layer Security Issues

Security at the Datalink Layer is generally restricted to encrypting the contents of frames since this layer is only concerned with a single hop on the network. This approach is acceptable when data is being transmitted over a point to point service such as a LAPB link when both sides know who the other side is and key distribution does not present a large administrative or technical overhead. In the case of local area networks however the problem of key distribution and proliferation among a potentially large number of network machines becomes a problem.

Another shortfall with encrypting at this layer is that all intermediate network layers must have access to the frame encryption key to allow routing information to be extracted. This potentially compromises the user information in the frame.

3.2.2.1. End to End Encryption

In the point to point case, security of a network can be enhanced by the placement of an encryption device as part of the sending and receiving modems or NTUs. Several commercial types of such devices exist and are readily available.

3.2.3. Network Layer Security Issues

As with the Datalink Layer, Network Layer security tends to involve encrypting network packets. In this case however host to host encryption can be established as network header information can be left in cleartext to allow the passage of a packet through the network. Within the TCP/IP Internet suite, the IP protocol includes provision for security extensions but lacks some of the key management and source authentication details required. A proposed protocol known as **swIPe** [19] addresses these issues and provides a mechanism to secure networks at this layer.

These are several disadvantages in using a mechanism that encrypts data packets. Since encryption occurs at a relatively low level of the protocol stack the user is faced with the situation where higher layers of the stack must process user data "in clear" prior to reaching

the Network Layer. Given that most implementations of higher level protocols (e.g. TCP and UDP) have facilities to trace and monitor data going through their respective layers the user must place considerable trust in the implementation of these layers as well as users that may have access to them (either as administrators or otherwise).

Generally Network Layer services (e.g. IP and X.25) provide a connectionless service between two hosts. This implies that packets may be lost, duplicated and/or arrive out of order. These problems must be sorted out by higher layers of the protocol. When encryption is used care must be taken that all encryption activity on individual packets is independent of any other packet.

3.2.3.1. swIPe

The swIPe protocol provides an extension to the TCP/IP Internet suite with three additional security services.

- Data confidentiality: Protection against unauthorised access to data being transmitted.
- Data integrity: Protection against alteration or replaying of traffic.
- Source Authentication: Network addresses are authenticated as part of the protocol.

SwIPe consists of three conceptual areas:

- A Policy engine which determines the action to be taken on outgoing packets, whether to accept incoming packets and how to process accepted incoming packets.
- A Key management engine which handles the production and distribution of keys. Master keys are dealt with on a manual basis.
- The Security engine which does the actual work of encryption, authentication, integrity and confidentiality checking under the direction of the Policy engine.

SwIPe packets are enclosed within IP packets using the IPIP extension to IP (IPIP - IP inside IP Protocol). The contents of the packets are encrypted using DES with MD5 used for authentication.

In operation, a packet to be transmitted is passed to swIPe where the policy engine decides whether authentication and/or encryption is required. If so, then the key management engine provides appropriate keys and the security engine does the required encryption/authentication passing the packet on to IP for transmission to the required host. On reception the receiving host takes roughly the opposite path to deliver clear data to the user.

Since IP does not provide a reliable delivery service the keys used to transmit packets belonging to the same session must be fixed. This eliminates the possibility of using any form of feedback stream cipher in the operation of this protocol. Similarly all packets must be padded to the minimum block size for encryption thus wasting potential bandwidth.

3.2.4. Transport Layer Security Issues

The Transport Layer provides a convenient place to segregate an organisations network from the outside world or other networks within the organisation. The boundaries of these networks are equipped with gateways to provide services such as protocol and media conversion, interfacing to telecommunication carriers and network security. When a gateway is used at a network boundary to provide security features it is often called a *firewall*.

As with the Network Layer it is possible to provide an end to end encryption facility between two "networks" over a hostile internet. In contrast to the Network Layer the Transport Layer provides a reliable message passing service thus allowing the use of more sophisticated feedback stream ciphers instead of the fixed key type arrangement used by swIPe.

3.2.4.1. Firewalls

The objective of a firewall is to prevent unauthorised access to the "insides" of a protected network by a variety of means. Within TCP/IP networks it is possible to establish gateways that discriminate on network traffic based on source and/or destination IP addresses thus only allowing known hosts to connect into or out off the protected area of the network. It is

also possible to limit access to certain TCP ports, thus only allowing certain services that use well known addresses such as telnet or ftp, etc to function through certain hosts. These restrictions can include running certain services such as mail, domain name server and time servers on only trusted hosts within the network and thus funneling all such functions to and from them in a controlled fashion.

Monitoring functions can be performed on such firewalls by including well known "wrapper" software which intercepts and records requests of a suspicious nature.

3.2.5. Session Layer Security Issues

At the Session Layer it is possible, for the first time in the protocol stack, to differentiate between individual users in a meaningful way. To this point, only system generated or stored keys have been used for encryption purposes. It is now possible and convenient to obtain keys directly from users on a use by use basis. This allows for the implementation of features which include such things as one time use scratch keys and challenge response mode protocols.

Some of these concepts have been implemented in secured versions of telnet and ftp.

3.2.5.1. Secure telnet/ftp

Although telnet and ftp protocols have been described as belonging to the Presentation Level services, the extensions proposed by Brown [12,13,14] can loosely be considered "Session Level" since they occur below Presentation services such as file transfer and screen interpretation.

In the proposed protocol, telnet and ftp sessions can include a challenge response sequence that allows a host to authenticate a potential user from a remote and unsecured network by passing a 64bit random number to the user and verifying that the response is appropriate. The reverse can also be achieved. Having authenticated that the user/host is who they said they were the pair can now exchange data using either the DES or LOKI encryption schemes. These facilities can be turned on or off at any time.

3.2.6. Presentation Layer Security Issues

Little implementation or standardisation effort has been placed into the general area of Presentation Level services with most services being provided on a ad-hoc basis. One attempt to provide such services in the area of security comes from AT&T which has developed a grab bag of library routines to assist in the provision of general purpose encryption tools.

3.2.6.1. CryptoLib

CryptoLib [26] provides a collection of routines that can be used as addin services when developing secure network protocols. The library is written in C and provides a portable and efficient method of supporting public and private key systems. The advantage of having such a library at this level of the protocol stack (as opposed to the Application Layer) is that each application does not have to rebuild the functionality provided by the library. Facilities provided include:

- Big integer creation and manipulation tools
- Big Arithmetic functions: addition subtraction, division, multiplication and shifting.
- Modular exponentiation and multiplication and the modulo operation.
- Bitwise AND,OR and XOR
- Chinese Remainder Theorem speedup of exponentiation
- Quadratic residue test, square root (mod prime), square root (mod composite of two primes)
- Random number generators - pseudo and true
- Prime number generators and Primality test
- Euclid's Extended Greatest Common Divisor algorithm
- DES encryption and decryption
- RSA methods
- El Gamal methods

- NIST key generation and DSA
- NIST Secure Hash Standard (SHS)

3.2.7. Application Layer Security Issues

At the application level, security of services is required. These must be adapted to take into account the type of service being offered and the level of security required. The price of this flexibility is that security related issues must be provided for every application on a case by case basis.

Two examples of security enhancements at the Application Layer are provided below. The first deals with enhancements for electronic mail and the second with a distributed authentication scheme developed as part of Project Athena [36].

3.2.7.1. PEM

PEM or Privacy Enhanced Mail [8,23,25] provides electronic mail users with security enhanced procedures for use over the Internet. It uses both public and private key encryption methods and supports a key management procedure as part of its protocol.

The services provided by PEM include confidentiality, authentication, message integrity assurance and non-repudiation of origin. Interoperability is provided by making use of the existing textual mail environment provided by SMTP (RFC822) by translating encrypted messages into a canonical format prior to using the underlying mail transport facilities.

The PEM system applies security prior to any underlying transport system and can be regarded as a true end to end security enhancement.

While PEM provides some enhanced services, other are not provided. These include access control, traffic flow confidentiality, address list accuracy, routing control, other issues relating to the reuse of serials by multiple users, assurance of message receipt and non deniability of receipt, automatic association of acknowledgement with original messages and message duplication and replay prevention.

PEM works by encapsulating an existing mail message with well defined headers specifying the format and key retrieval information required for reading/authenticating the message. A message in such format is translated into canonical format (printable characters) acceptable to SMTP. On delivery the receiving PEM uses the header information along with key information provided by the receiver to decode/authenticate the message.

3.2.7.2. Kerberos

Kerberos [36] provides trusted third party authentication services in a distributed network environment. It does this by maintaining a database of clients and their associated private keys and using these keys to convince one client that another is really who they say they are.

Once authentication has been established Kerberos can then issue session keys that have a limited life span. These keys are called tickets and have a one to one relationship between a client and a server. Once a ticket expires a new ticket must be obtained to gain access to a service again.

The ticket issuing function can also be delegated to a Ticket Generating Server which is separate from the original Kerberos authenticator. The database of clients is generally maintained in a read only fashion and updated offline to reduce the possibility of compromise.

The implementation of Kerberos requires the modification of many standard operating system facilities such as *login* and the BSD 'r' commands so that they use the Kerberos authentication functions.

3.3. TCP

Having considering networking in general, specific issues concerning the Transmission Control Protocol (TCP) [30] are discussed below.

TCP is one of a number of protocols originating from the TCP/IP Internet Protocol Suite that comes from research funded by the Defence Advanced Research Project Agency (DARPA). This suite contains many communication protocols that can be used as the basis for building

large connected networks (internets) that are able to run on a myriad of host platforms over diverse interconnecting network hardware and services.

As mentioned above, the TCP protocol sits above some underlying network delivery protocol and has been designed to provide a reliable stream delivery service over a potentially unreliable network. The relevant position of TCP and some protocols directly related to it is shown below.

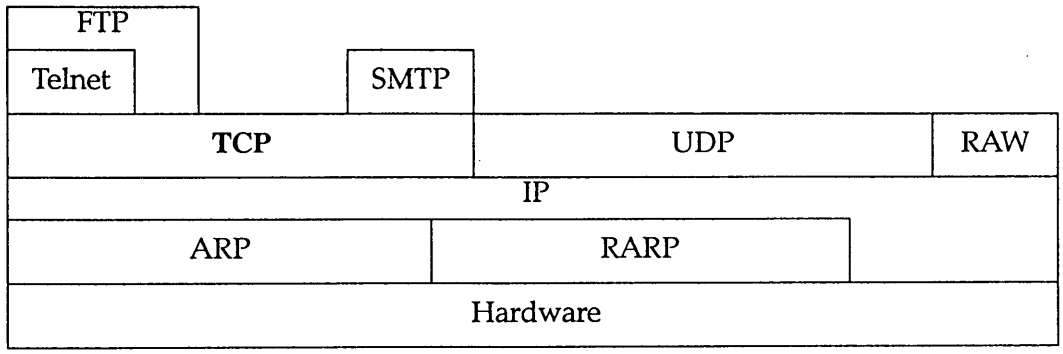


Fig #9 - TCP's position in the network structure

With the goal of reliability established TCP provides a number of features that characterise the protocol.

The TCP protocol is a byte oriented stream with application programs transferring data in blocks of 8-bit bytes making up arbitrary chunks of data. The data formed in such streams is considered to be without structure or format and it is up to the application to manage or impose any format at some higher level.

The TCP protocol provides a Virtual Circuit Connection. Two processes communicate through a conceptual scheme similar to the placement of a telephone call. In this scheme there are methods of connection (calling) and accepting connections (pickup or answer), when connections have been established communication can take place until one or both parties decide to finish a connection (hangup). In the case of errors or unexpected

disconnections, methods are provided to inform the other party that the connection has been discontinued.

TCP also provides buffering to improve the efficiency of use of any underlying network. An application program may send single octets of data at a time or huge packets but the TCP protocol buffers, disassembles or assembles packets or messages then multiplexes transmission over the network layer of the software. At the other end of the network TCP will demultiplex packets as required and deliver them to the application in the same order that they were sent.

TCP provides full duplex connection so that applications can send and receive data on the same connection "at the same time".

The area of TCP that is of particular interest is the method of establishing sessions or connections between co-operating processes. It is at the point of establishing connections that it is most convenient to establish encryption. The distinction is made between the architecture of establishing a connection, the formats and protocols of messages involved in making a connection and the implementation details of making a connection. These three differing levels have an impact on what is possible within the scope of TCP.

3.3.1. The architecture of a TCP connection

The architecture of a connection refers to a high level description of the activities of establishing a connection. It does not concern itself with the details of the meaning of bit patterns within a frame or packet.

The basic architecture of a TCP connection is a three way handshake. This provides the level of reliability needed for a full duplex connection oriented protocol where it may happen that two processes decide to initiate a connection with each other simultaneously. The three way handshake is designed to provide correct synchronisation regardless of the underlying delivery mechanism and regardless of the order of actions taken by processes attempting to establish a connection. When two processes are about to establish a connection we can call

one the client and one the server. Without loss of generality the client can be said to initiate the session when it sends a SYN (*synchronisation*) message to the server. The client enters a state known as SYN-SENT and waits. The server will, at some time in the future, receive a SYN message and will respond by sending a SYN message with an ACK (*acknowledge*) attached. The server will then enter a state known as SYN-RECEIVED. At some later time the client will receive the SYN-ACK message and will repond with a plain ACK message. The client will then enter the ESTABLISHED state. The server then receives the ACK message and enters the ESTABLISHED state. Thus a connection is established by three way handshake.

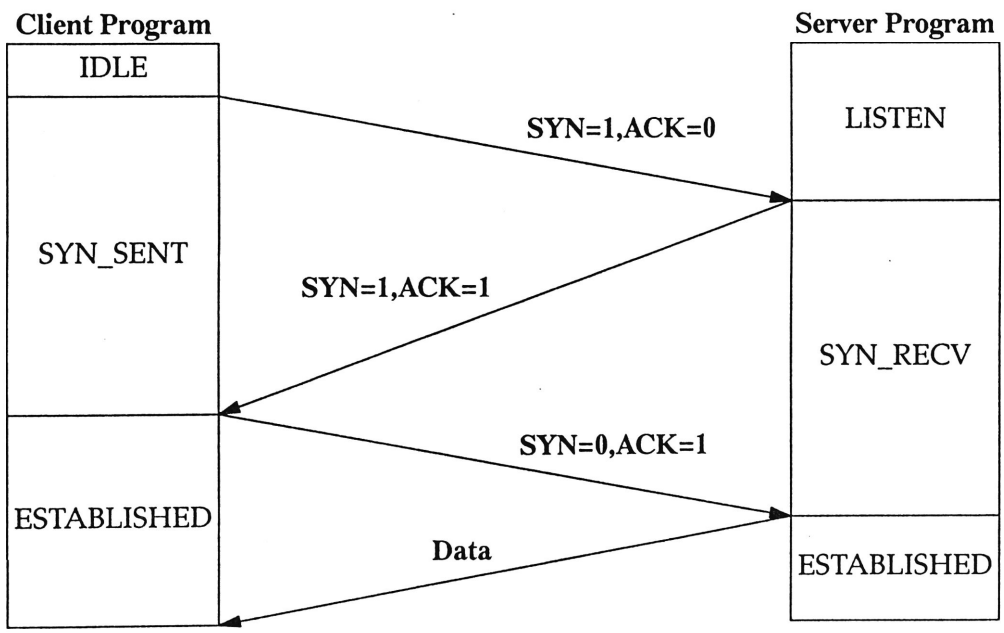


Fig #8 - Establishing a connection

The diagram shows the progress of a three way handshake. Time travels down the page and messages of appropriate type are indicated by arrows. State transitions are shown along the respective sides of the client and server.

3.4. Formats and Protocols of TCP

Formats and protocols give meaning and semantics to bit patterns within data packets used to express the architecture of TCP. This is the case with both TCP and its associated IP protocol. The basic structure of the TCP message is a TCP header followed by arbitrary data. The header is padded to a 32 bit boundary as is the data.

The header area contain a minimum of five 32 bit words with a format shown below.

Format of a TCP Header

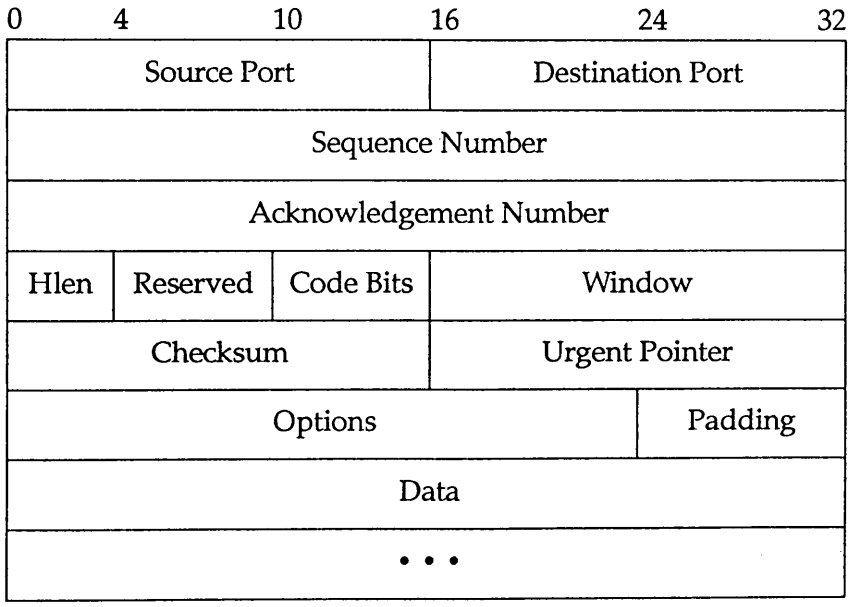


Fig #9

The source and destination ports identify the end points of the connection. Host addresses are dealt with by the IP Protocol which encapsulates the TCP message. The sequence number identifies the place in the input stream where the data in this packet belongs. The acknowledgement number identifies the next position expected to be received. Hlen is the number of 32 bit words in this header and is used as an offset to find the beginning of the data. Code Bits indicate whether the current message contains urgent data, whether this message has a valid acknowledgement field i.e. is an ACK, whether data on this connection was pushed i.e. buffers explicitly flushed by application request, whether we want to reset

the connection i.e. hangup, whether this is a SYN message and whether this is the last message of this connection. The window field indicates how much buffer space is available on the sending host. This field can be used to restrict sending data when there is no buffer to save it. Checksum provides a method of verifying the integrity of the header. The urgent pointer (when valid) indicates the offset of out of band data (Out of band data is discussed below).

3.4.1. Option fields

Following the urgent pointer we find the optional option fields. Option fields provide TCP with a mechanism to pass relatively small arbitrary messages outside the standard data stream. The original TCP protocol standard defines the format of option lists as follows.

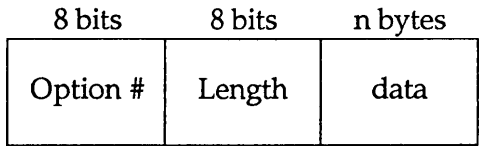


Fig #10 - Format for Option fields

Where "Option #" is an option number between 0 and 255. Length provides the number of bytes that the total option field contains (including the Option # and the Length). The data can be arbitrary data associated with the option. The TCP standard defines three option numbers:

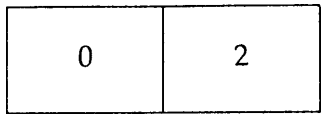


Fig #11 - End of Options

Option 0 is the end of option list option. This option can be used to terminate a list of multiple option fields in an option list. Note the length field is two bytes, these being the option number and the length field itself.

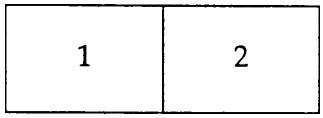


Fig #12 - Nop Option

Option 1 is the nop option. This option is a no operation option and is simply a place holder in an option list. Again the length field is two bytes.

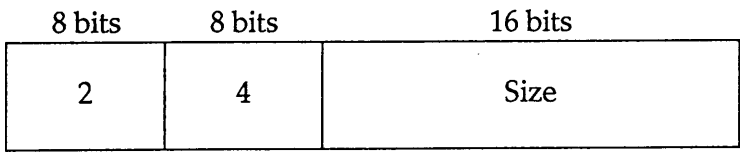


Fig #13 - MSS Option

Option 2 is the Maximum Segment Size (MSS) which allows a TCP to set the maximum size of a message that it is willing to accept.

All other option are undefined but available for future use. Option fields will be used to carry indications that encrypted data streams are in use and this is discussed at length below.

3.4.2. Data

Data consists of an arbitrary number of bytes. Each byte is assigned a conceptual sequence number that defines its position in the datastream.

Typically data is delivered in the order prescribed by the sequence numbers. There is, however, a facility that allows data to be delivered out of this strict sequence. The so called Urgent Pointer gives the position of such "out of sequence" data and one of the Code Bits indicates the existence of the urgent data. When a TCP receives a message with urgent data it immediately delivers this data to the application, bypassing any data that is expected or already buffered.

3.5. Implementation of a connection in TCP

The details of the implementation of TCP are dependent on the method of delivery. A TCP can reside in a variety of hardware and software platforms. These range from special purpose routers and protocol converters to general purpose hosts. In a host based environment the TCP can be part of the kernel or part of the user's address space depending on the type of host. The particular implementation discussed below runs TCP as part of a kernel on an inexpensive multiuser host.

4. Linux

Linux is a POSIX compliant implementation of UNIX written for inexpensive multiuser hosts, predominantly Intel 80486 based microprocessors. The Linux project was started to provide a shareware version of UNIX with full source code distribution. Various utilities, compliers and tools from the GNU project are used to build and maintain the system. Currently the Linux operating system is selfsupporting, i.e. support and development of Linux can be undertaken from within a Linux environment. Support is provided for a range of hardware including video adapters, disks, CD-ROMS, sound cards, modems, tape drives. The implementation used and described below consists of an Osborne 80486 based PC with a 240MB Quantum Fixed Disk and VD24X Diamond Video Adapter running Linux Version 1.0.8, gcc version 2.5.8, X11R5 and Cornell tcsh 6.04.

4.1. Linux module organisation

Linux is organised in a hierarchy of directories that reflect the modular design of the system. Typically system source code is located in a directory called *linux*. This *linux* directory is typically located in */usr/src* but this need not be so.

The operating system module libraries are then arranged in directories according to their subsystem functions. Such functions include *lib* that is used to hold kernel libraries, *fs* which hold file system code etc. All network related code is held in the *net* directory. This directory holds the highest level interface code for the socket calls made by user programs. The *net* directory is further broken down into subdirectories containing implementations of the various domain types. Among the domain types is Internet which is found in the *inet* directory. This contains the source code for the entire internet suite as well as the extension proposed by this paper.

At execution time, linkage between the highest levels of the socket interface and the specific implementations is provides by blocks of function pointers which are initialised dynamically at kernel startup time with domain initialisation routines.

A small part of the directory structure of Linux is shown in the diagram below.

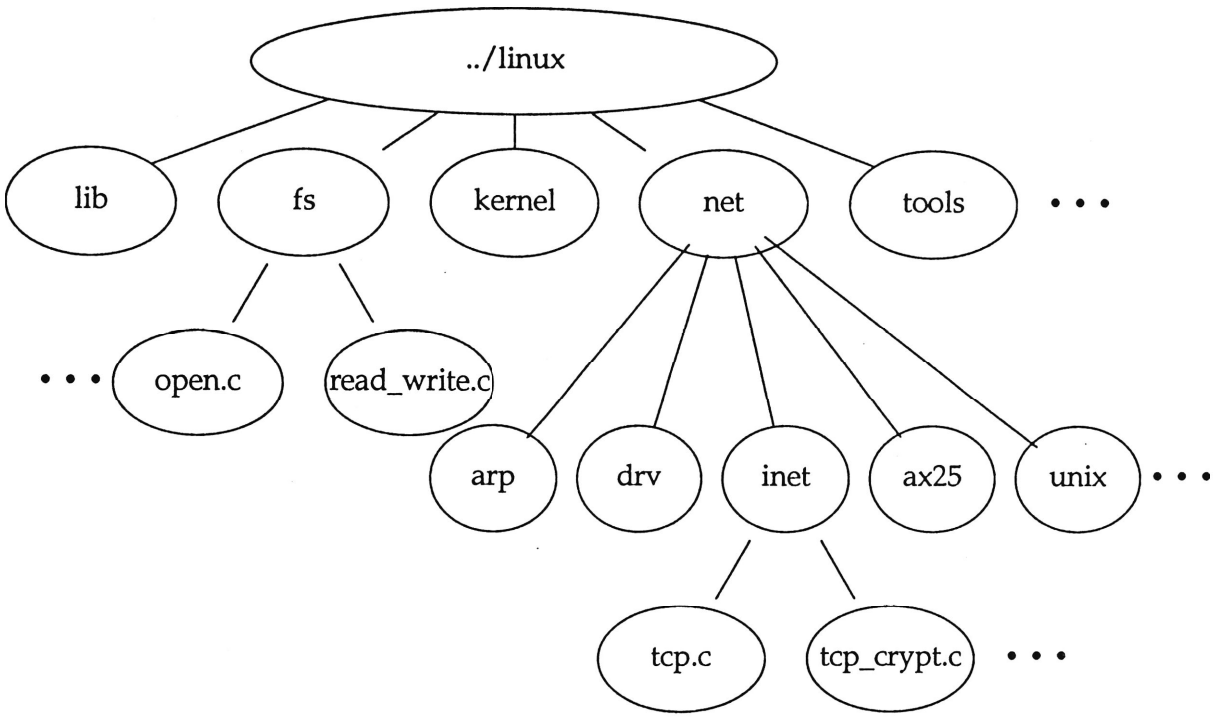


Fig #14 - Linux source code directory

4.2. A tour of the Linux Networking Kernel Code

4.2.1. The client

We consider the actions that take place inside the kernel as a result of the execution of two simple client-server programs. The client program shown below opens an internet socket stream resulting in the return of a file descriptor (*sock*), initialises a data structure (*sin*) with values specifying the host and port that will be connected to, it connects the socket to that host/port then reads what is sent and prints it. Finally it closes the file descriptor (*sock*) and exits.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "c-s.h"
#define BUFLEN 128
main(argc,argv)
int argc;
char *argv[];
{
        int sock,n;
        struct sockaddr_in sin;
        unsigned char buf[BUFLEN];

        sock = socket(AF_INET,SOCK_STREAM,0);
        if ( sock == -1 ) {
                fprintf(stderr,"Socket returned -1\n");
                exit(-1);
        }
        bzero((char *)&sin,sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr =  inet_addr(MYIP);
        sin.sin_port = htons(MYPORT);
        if ( connect(sock,(struct sockaddr *)&sin,sizeof(sin)) < 0 ) {
                fprintf(stderr,"could not connect\n");
                exit(-1);
        }
        while (( n = read (sock,buf,BUFLEN) ) > 0 ) {
                buf[n] = '\0';
                puts(buf);
        }
        fputs("\n",stdout);
        close(sock);
        exit(0);
}
```

Fig #15 - A simple client program

4.2.2. The server

The server program provides data that is read by the client. Firstly the server creates an internet socket stream and assigns it to a file descriptor (*sock*), the server then initialises a structure for holding information about future communication links (*sin*). The values in this *sin* structure indicate that it will be an internet connection accepting connection on any

incoming address that this host may be known as and on a specific TCP port number (*MYPORT*). The socket (*sock*) and the address structure (*sin*) are then bound with the *bind* call and the file descriptor is set into a passive state with the *listen* call, at some time in the future a new file descriptor (*wsock*) is created when the client and server programs rendezvous via the clients *connect*. The server writes a message to the newly created file descriptor (*wsock*), closes its files and exits. The total result of the above is that the message "Hello World" is printed by the client.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>
#include "c-s.h"
main(argc,argv)
int argc;
char *argv[];
{
        int sock,wsock,len;
        struct sockaddr_in sin,fsin;
        unsigned char pch[] = "Hello World";

        sock = socket(AF_INET,SOCK_STREAM,0);
        if ( sock == -1 ) {
                fprintf(stderr,"Socket returned -1\n");
                exit(-1);
        }
        bzero((char *)&sin,sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_port = htons(MYPORT);

        if (bind(sock,(struct sockaddr *)&sin,sizeof(sin)) == -1 ) {
                fprintf(stderr,"bind returned an error\n");
                exit(-1);
        }
        if ( listen(sock,5) < 0 ) {
                fprintf(stderr,"listen could not listen?\n");
                exit(-1);
        }
        wsock = accept(sock,(struct sockaddr *)&fsin,&len);
        if ( wsock < 0 ) {
                fprintf(stderr,"Accept failed\n");
                exit(-1);
        }
        write(wsock,pch,strlen(pch));
        close(wsock);
        close(sock);
        exit(0);
}
```

Fig #16 - A simple server program

We now consider the view from the kernel's side which is considerably more complex. The server is considered first as the client would exit with an error if the server was not ready to

accept connections.

4.2.3. The Server

4.2.3.1. *socket*

On making the call to *socket* the server enters the kernel through the *_system_call* entry point in linux/kernel/sys_call.S. The system routine to be executed is held in the eax register of the CPU and is used as an offset into a table (*_sys_call_table*) which forms the address that is then called by the *80486 call* instruction. This call leads to the execution of the *sys_socketcall* function in linux/net/socket.c

The parameters to the *sys_socketcall* routine are the type of socket call (socket, bind, connect ... etc) which is encoded as an integer and the arguments relevant to the call. This allows a simple switch statement to be used to call the relevant routine. In this case *sock_socket* which is also located in linux/net/socket.c.

The *sock_socket* routine takes as an argument the familiar family, type and protocol parameters of the application program *socket* call. It firstly searches the kernel's domain protocol table until it finds a matching family, which in this case will be the *AF_INET* domain. The protocol domain structure contains the necessary linkages between families of domains and their underlying subroutines. Having found the family of domains it then finds a free socket structure from the *sockets* array (defined in this file) by calling the subroutine *sock_alloc*.

The *socket* structure which is returned from the *sock_alloc* is then partially initialised and completes its creation with a call through a pointer to a create function held in the domain protocol structure. In this case the called routine is *inet_create* in linux/net/inet/sock.c. The call to this routine represents the first real transition between protocol independent code and protocol specific code.

The *inet_create* allocates a new structure called a *sock* to hold information necessary to maintain a *AF_INET* connection. It initialises this structure and links the necessary TCP

routines through another type of protocol structure specific to *AF_INET* domains. It then links this *sock* into the *socket* structure allocated above in *sock_socket*. Finally the *inet_create* routine calls a protocol specific initialisation routine, which for *AF_INET* domains is a null routine.

Now we return to *sock_socket* and allocate an operating system file descriptor through the routine *get_fd* in linux/net/socket.c. This ensures that calls to *read* and *write* will be subsequently passed on to the appropriate handler routines. *Sock_socket* now returns the file descriptor to *sys_socket* which in turn returns to the user through the *system_call* gateway.

4.2.3.2. *bind*

The *bind* system call ends up at *sys_socketcall* following the same path as *socket*. At this point the switch statement takes us to *sock_bind* in linux/net/socket.c which simply ensures that the file descriptor passed as a parameter is valid and calls the underlying bind routine *inet_bind*.

The *inet_bind* routine does some error checks and then copies the requested Internet TCP address structure from users address space into the relevant internal tables and sets some parameters where default values have not been specified. It then places the socket into the domain socket array with its appropriate port number.

By the time the *bind* call has completed data structures in the kernel have been established from the file descriptor *fd* through to the actual TCP specific parameters held in the *sock* structure. This is shown below.

The *current* pointer gives the location in the kernel's task structure array of the currently active process. The *filp* array is the array of file descriptors within the task structure of a process.

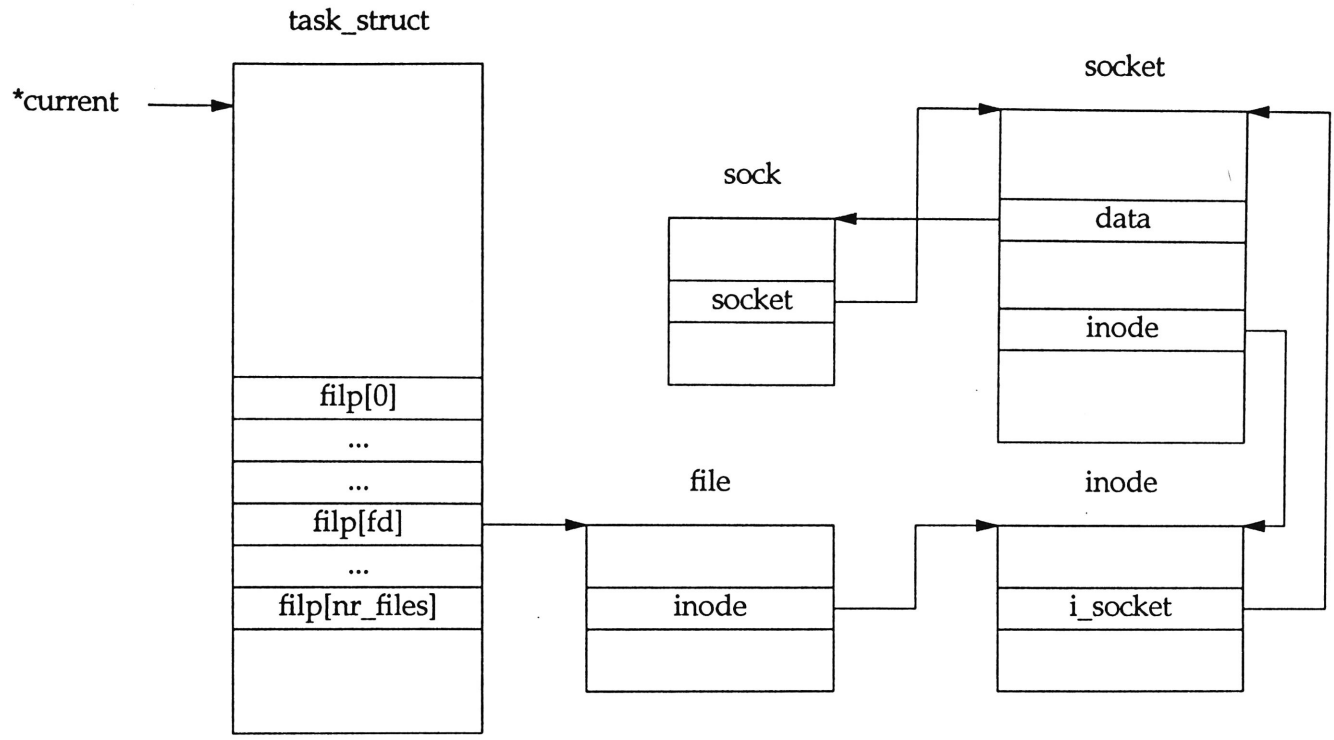


Fig #17 - Data Structures within the kernel after a *bind* call.

4.2.3.3. *listen*

The *listen* calls ends up at *sys_socketcall* following the same path as *socket*. At this point the switch statement takes us to *sock_listen* in linux/net/socket.c which uses the file descriptor passed as a parameter to look up the kernel *file* pointer associated with this descriptor by calling *sockfd_lookup* in linux/net/socket.c. Having found the *file* it then looks up the *inode* associated with the *file* using the *socki_lookup* routine also in linux/net/socket.c Finally *sock_listen* calls *inet_listen* to initialise fields associated with the *sock* structure and sets a flag in the *socket* structure to accept connections.

4.2.3.4. *accept*

The *accept* call ends up at *sys_socketcall* following the same path as *socket*. At this point the switch statement takes us to *sock_accept*. *sock_accept* uses the file descriptor to return the kernel *file* using *sockfd_lookup* in linux/net/socket.c which in turn uses the *file* and

socki_lookup in linux/net/socket.c to obtain the *inode* which points to the *socket*. The state of the *socket* is checked to ensure that it is in an appropriate state to be used in an *accept* call and then a new *socket* is allocated with *sock_alloc*. Having obtained the *socket*, its contents are now initialised by using the *inet_dup* routine in linux/net/socket.c which simply calls *inet_create* using the original socket's *sock protocol*.

The *AF_INET* specific accept routine *inet_accept* in linux/net/socket.c is now called. *inet_accept* starts by deallocating any storage allocated to the *data* field in the new socket. It then calls the TCP specific *tcp_accept* in linux/net/inet/tcp.c which waits until a packet comes for it using a loop on the *get_firstr* from linux/net/inet/skbuff.c. When the buffer arrives the *sock* associated with this buffer is returned. This returns us to *inet_accept* which assigns the returned *sock* to its *socket*, sets the TCP state to TCP_SYN_RECV and waits to establish a connection in the TCP_ESTABLISHED state by the usual TCP threeway handshaking method.

At the end of this we have a new socket connection established with the existing process. This is used to carry further messages.

4.2.3.5. *write*

The *write* call enters the kernel though the usual *_system_call* entry point in linux/kernel/sys_call.S. This leads to a call to *sys_write* in linux/fs/read_write.c which does a few error checks and then does an indirect call though a pointer to *sock_write*. *sock_write* in linux/net/socket.c calls *inet_write* in linux/net/inet/sock.c which calls *tcp_write* though the protocol table associated with this socket. *tcp_write* in linux/net/inet/tcp.c loops until all of the data to be sent has been processed. Inside the loop *tcp_write* either creates a buffer to send and sends it or creates a partial buffer and queues that partial buffer. There are two places in the code where the *memcpy_fromfs* routine is called to fetch user data from the user address space, one in the area where complete buffers are sent and the other where partial buffers are built.

4.2.3.6. *close*

The *close* call enters the kernel though the usual *_system_call* entry point in linux/kernel/sys_call.S. This leads to a call to *sys_close* in linux/fs/open.c. *sys_close* calls *close_fp* in linux/fs/open.c which calls *sock_close* though the familiar function protocol pointer. *sock_close* in linux/net/inet/sock.c looks up the inode associated with this file and calls *sock_release* in linux/net/socket.c. *sock_release* calls *inet_release* in linux/net/inet/sock.c which in turn calls *tcp_close* in linux/net/inet/tcp.c which goes through the TCP handshaking shutdown required as part of the TCP protocol.

4.2.4. The Client

4.2.4.1. *socket*

The operation of *socket* in the client program is the same as the server program.

4.2.4.2. *connect*

The *connect* enters the kernel in the same way as *socket* and other socket based calls and really starts its journey through *sock_connect* in linux/net/socket.c. *sock_connect* calls *inet_connect* in linux/net/inet/sock.c which allocates a socket number if required and calls the *tcp_connect* routine in linux/net/inet/tcp.c which builds a header with the SYN bit set and transmits it to the desired host via IP. Control is then passed back to *inet_connect* where the kernel sets up a *sleep_on_interrupt* state for this process and loops, waiting for either a timeout or a response to the connect which comes from *tcp_rcv* (described below).

4.2.4.3. *read*

The *read* call enters the kernel though the usual *_system_call* entry point in linux/kernel/sys_call.S. This leads to a call to *sys_read* in linux/fs/read_write.c which does a few error checks and then does an indirect call though a pointer to *sock_read*.

sock_read in linux/net/socket.c looks up the *socket* associated with this *file* and then calls *inet_read*. *inet_read* in linux/net/inet/sock.c binds the socket if required then calls *tcp_read*.

tcp_read in linux/net/inet/tcp.c deals with the complexity of reassembling incoming IP packets which may arrive out of sequence, duplicated or which may be entirely lost. The *tcp_read* routine starts by checking the consistency of calling parameters, it then checks whether we are attempting to receive urgent data. In the case of *inet_read* we are never attempting to read out of band data. If no data is available in the receive queue and no other errors have occurred the kernel reschedules and we resume when data becomes available. If data is currently available then it is removed from the receive queue and copied to the users address space. The amount of data available and copied is returned to the user to indicate success.

4.2.4.4. *tcp_rcv*

Although never called explicitly by the user, *tcp_rcv* in linux/net/inet/tcp.c is an important routine in the overall scheme of things. *tcp_rcv* is called by *ip_rcv* in linux/net/inet/ip.c as the routine that handles an incoming IP datagram that is designated for TCP. *tcp_rcv* finds the socket that the packet belongs to by using the port, address and socket number and, for normal data, calls *tcp_data* to place this data on the socket's receive queue. This is how *tcp_read* can reschedule and return to find data has appeared on the receive queue. The *tcp_rcv* routine also takes special action when a packet that requests a new connection arrives. After determining that a new connection is being requested, *tcp_rcv* calls the *tcp_conn_request* routine in linux/net/inet/tcp.c which takes the required steps to rendezvous with a previously "listened" socket which is "accepting".

4.2.4.5. *close*

The operation of *close* in the client program is the same as the server program.

5. Design

The general design of the proposed enhancement is now discussed.

5.1. Design Assumptions

It is intended that an enhancement to network security be provided. Any enhancements will require design decisions to be made regarding the architecture, protocols and implementation of the system. Assumptions and alternatives need to be weighed up when making these decisions.

The first major assumption used to drive the decision making process for this enhancement is that the proposal will in no way "break" the existing architecture, protocols or operation of an existing (correct) implementation of TCP/IP. This is a desirable assumption as interoperability between existing implementations is critical if this enhancement is to be used in an environment where not all platforms have access to the enhancement.

The second major assumption is that control and security may be maintained by either or both of the operating system kernel or end user application, depending on configuration and initialisation. This assumption allows greater flexibility in the use and deployment of this enhancement.

The third major assumption is that TCP is the appropriate place to put a security mechanism. This will be discussed shortly.

These assumptions will be tied to the design decisions made in the following paragraphs.

5.2. The Architecture of the Proposed Enhancement

The proposed enhancement consists of a conceptual layer above the TCP protocol in the network implementation stack. The layer is implemented largely as a separate module within the operating system kernel and is controlled by parameters set within the kernel at startup time and subsequently dynamically updated at runtime.

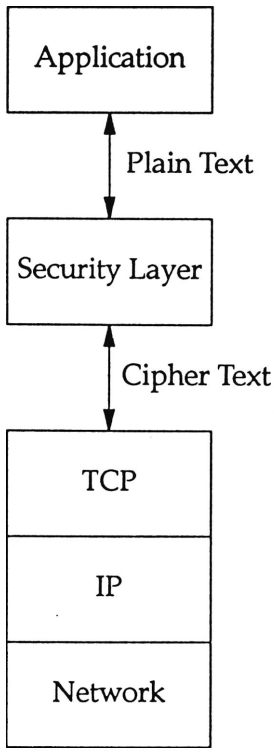


Fig #18 - Security Layer in the Network Protocol Stack

The basic architecture of the security layer is shown below. It consists of four modules: key exchange, master keys, session keys and the encryption components.

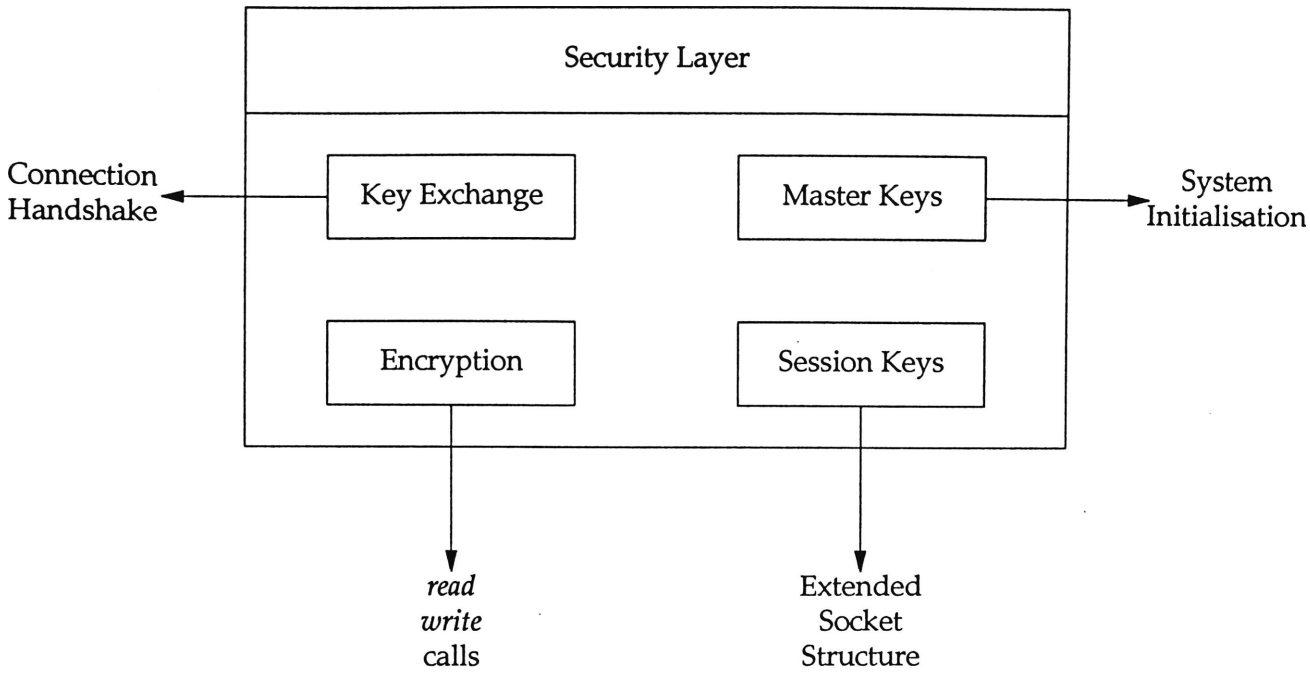


Fig #19 - Functional Security Layer Modules

5.2.1. Master Keys

Master keys play an important role in the application of encryption to TCP. Each implementation of TCP must be able to establish secure communication with a known and trusted partner. Several mechanisms are available to establish this trust but at the end of any such mechanism there must be delivery of some key that two TCP's can agree on to encrypt data.

5.2.1.1. Purpose of Master Keys

The purpose of the master keys is to allow overall administration between pairs of TCP's. This two level approach to key management is similar to the way that we may use an asymmetric key exchange mechanism to exchange keys for a subsequent (faster) symmetric encryption operation. The keys used to implement the administrative functions can be unrelated to subsequent keys and need not even be the same encryption function.

The presence of a mechanism for storing and using master keys is required because the second design assumption calls for the ability of applications to provide keys for individual sessions. This would be impractical if a single level key directory was imposed.

5.2.1.2. Initialisation of Master Keys

The master keys are initialised for a TCP by a privileged kernel call which provides the address of the other partner as well as the key details. Simultaneously the other partner will register the same key details. This will usually be carried out at system initialisation time and it is assumed that master key values will change relatively infrequently compared to session keys.

5.2.1.3. Modification of Master Keys

Master keys can be modified by simply reinitialising them with the same kernel call that is used to initialise them, previous information for a given address will be over written.

5.2.1.4. Master Keys as the trigger for kernel initiated encryption

It is assumed that encryption may be initiated by either the end user application or the kernel. This implies that the default use of encryption is disabled, otherwise, the kernel would always initiate encryption, with the end user only overriding certain parameters. The "default non encryption" mode of operating also implies that there needs to be a trigger that will initiate user transparent encryption between TCP's. This trigger is held with the master keys. On initialisation, the master key for a host has a flag set indicating whether encryption shall be turned on by default, if this flag is set then appropriate fields will specify the style of encryption to be used along with some default initialisation vectors where appropriate.

5.2.2. Sessions Keys

Session keys provide the information required to drive the encryption process after session establishment.

5.2.2.1. Purpose of Session Keys

The purpose of the session key is to provide a unique key that can be shared between connected TCP sessions. The keys need to be unique (or at least random) because ultimately a socket belongs to a user process that shares an underlying communications transport mechanism that can be subject to interception. The association of a unique key with a unique socket pair provides the level of security that would be required by applications (as opposed to hosts or networks) that exchange secret data.

5.2.2.2. Storage of Session Keys

Session keys belong to a particular socket and are valid for the life of that socket. Two options are available to store the session keys. The actual socket structure within the kernel can be modified to include the key information or an additional structure can be added to hold the session keys. The second approach was favored as it reduced the impact and recompilation time on the kernel when changes were made to the session key structure. The session key structure is linked from the socket structure by a pointer, the memory associated with this extended socket arrangement is allocated and deallocated along with the socket structure at the appropriate time in the kernel.

5.2.2.3. Generation of Session Keys

Session keys can be generated in two ways, the kernel can generate a key or the end user can provide an explicit key that it wishes to exchange.

5.2.2.3.1. Kernel Generated Session Keys

When required, the kernel will generate keys based on a single internal register and the master key of the host to which it will exchange the keys. The register is updated using a modulo quadratic hash and this value is passed to the encryption module along with the master key of the host, the value returned by this encryption is then used as the session key.

The modulo quadratic hash provides some, but not a lot, of randomness in the generation of keys, the randomness and secrecy of the session keys is provided by the encryption of this hash by the master key of the host concerned. This reduces the chance that another host, that may know the quadratic hash sequence by establishment of some earlier session key, may be able to determine the value of a session key for another unauthorised host.

The kernel generated key model is show below:

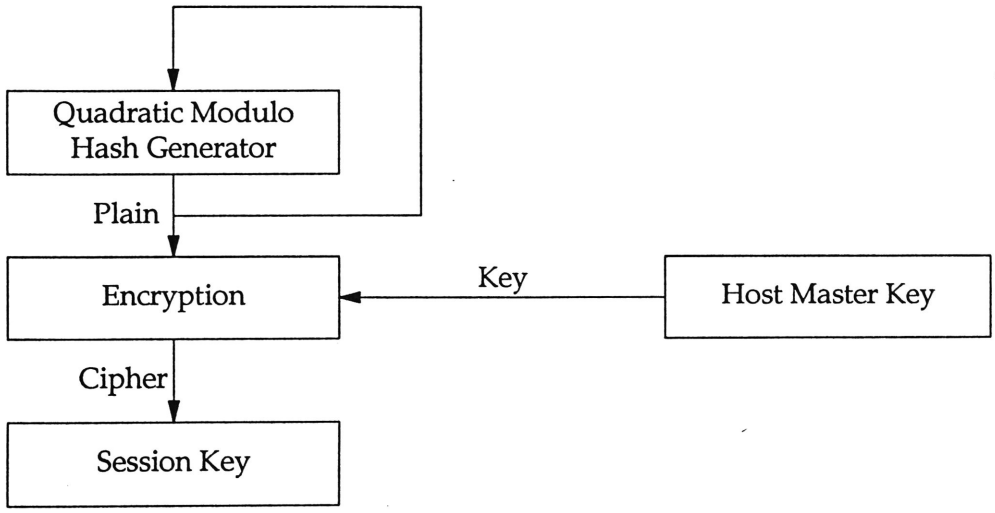


Fig #20 - Kernel Generated Session Keys

5.2.2.3.2. End User Generated Session Keys and Initialisation Vectors

As well as kernel generated keys, user applications can supply their own session keys. These are passed through enhanced kernel calls that are recognized by the security layer and passed onto the enhanced socket structure belonging to the socket in question.

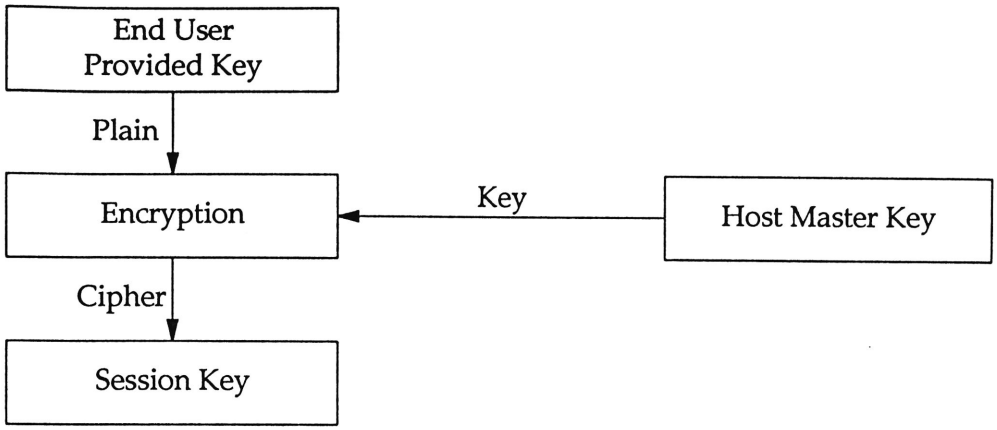


Fig #21 - User Supplied Session Keys

5.2.3. Key Exchange Mechanisms

The method of securely exchanging session keys is the most critical aspect of the proposed enhancement and requires careful decisions to be made regarding architecture. The general mechanism for exchanging keys is presented along with specific means of key exchange.

In discussing the architecture of key exchange it is assumed that we are actually referring to the exchange of encrypted session keys and the process of getting a new session key from one host to another is as follows:

- Generate Session Key. This is explained above.
- Encrypt the new session key with the appropriate master key.
- Pass the encrypted key to the partner.
- (Partner) decrypt the session key with the master key.
- (Partner) store the session key with the appropriate socket.
- (Both) use the session key to encrypt and decrypt data.

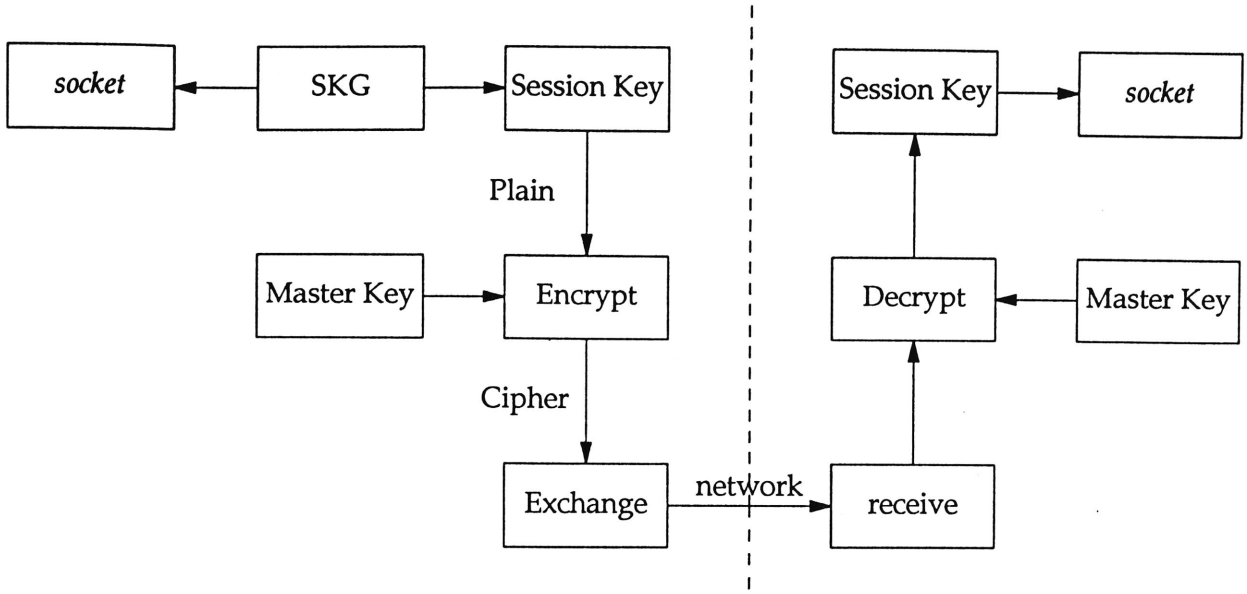


Fig #22 - Session Key Processing (SKG - Session Key Generation)

Several methods of actually conducting the exchange can be used. These are outlined below with some discussion of the relative merits of each style.

5.2.3.1. Using Spare bits in the TCP header

As outlined in section **3.4** there are a small number of unused bits within the TCP header that may be used to transmit key information. This method of exchanging keys is considered undesirable for a number of reasons. Firstly, it contravenes the assumption that this proposal will not break an existing implementation of TCP. RFC 793 [30] states that reserved bits must be zero. Using these bits for another purpose may result in a correct implementation of TCP rejecting theses headers. Secondly, if this method is used there are only a small number of spare bits available in a TCP header, typically six, although up to sixteen additional bits may by obtained when the smallest option list (also sixteen bits) is padded to its required thirty two bit alignment. This lack of available bits means that when faced with the requirement to deliver a sixty four bit or even one hundred and twenty eight bit key, multiple headers would be required to carry the key payload. Even with the three way handshake used to

establish a TCP connection it would be difficult to carry moderately large keys without substantial revision to the TCP protocol itself.

5.2.3.2. Inserting Key information instream

Another method of exchanging encrypted session keys is to embed the keys within the data stream using some recognisable escape sequence to detect them. This method would be suitable for an out of band key exchange mechanism (which is discussed below) but its goes against the general principle of separation of control information and data. This would mean that significantly less segregation of functionality could take place between the security layer and the TCP layer with one of these layers being responsible for the scanning of incoming data streams for possible key sequences and removing them and notifying the other layer of its operations. The requirement for one layer of a protocol stack to have intimate knowledge of the protocols, formats and operations of another layer largely goes against the general design goal of an OSI based protocol stack methodology.

5.2.3.3. Using an external mechanism

It is possible that some preexisting mechanism could be used to exchange encrypted session keys. This mechanism however, does not solve the problems of keys generation and the synchronisation of TCP's and its not in line with the architectural model shown in Fig 19

5.2.3.4. Using Option Lists

The final alternative for key exchange that is considered is through the option list mechanism. Option lists provide both IP and TCP a method of exchanging small amounts of useful non user data information between TCP's or IP based hosts. As seen previously, the TCP standard RFC 793 provides for the definition of three recognised option lists. This leaves considerable scope for the use of additional option definitions which have not been specifically reserved by the standard. Implementations of TCP ignore unrecognised options and so the delivery of a non standard option list should not cause a well behaved implementation of TCP to fail, merely pass on meaningless data to its application.

The maximum amount of data that can be delivered through an option list is considerably greater that the spare bits provided by the TCP header but not enough to be extravagant will key exchange ideas. There is insufficient room to pass both inbound and outbound keys and encryption initialisation vectors for algorithms that require one hundred and twenty eight bit keys but this can be dealt with as a two way handshake if required.

As a result of the above, the design decision was taken to use Option Lists as the mechanism to exchange keys.

5.2.3.5. In Band vs Out of Band Key Exchange

Two approaches can be adopted with the decision as to when it is appropriate to send keys between TCP's. Keys may be exchanged in band, that is, at a point in time that is synchronised with data transmission, or out of band, that is, a point in time that is not synchronised with data flow.

If an in band approach is used it seems reasonable that the appropriate point in the transmission to insert the key exchange is prior to any data being transmitted. The disadvantage is that flexibility is lost and the option to either discontinue or modify the mode of operation is also not available. It is also difficult to implement an in band scheme that uses other than the period prior to data flow to exchange keys without radically altering the architecture of TCP.

The second approach to key exchange is to use an out of band solution. This can present potential problems with synchronising data when we are not sure what application layer programs above in the protocol stack are doing. Particularly if race conditions develop when applications consume data before a key exchange can take place.

For this final reason an in band key exchange solution adopted with the key exchange being executed prior to data transmission.

5.2.4. Encryption

The encryption module has two major functions. Firstly to carry out encryption of data streams where required and secondly to initiate key exchange and session key generation as explained above.

5.2.4.1. Operation

As seen previously, application programs that wish to communicate with TCP do so via the various network system calls available (*socket, connect, read, write, close*). Placed between these calls and the actual TCP layer is a decision module which will determine if security related activity will take place. This is implemented as a flag associated with socket structure and will be be set if security is needed otherwise it will not be set.

If security related activity is to happen *read/write* calls will be intercepted by the security layer and actions taken to encrypt/decrypt data as needed. When these actions are completed data is either encrypted and passed down to the TCP layer (for a *write* type call) or decrypted and passed up to the application (for a *read* type call).

Also, if security related activity is to happen various socket setup calls will be intercepted and action taken to either swap user provided keys or generate and swap kernel generated session keys.

5.2.4.2. Variable Encryption methods

Part of the key exchange process involves selecting the encryption scheme to be used during encryption operations.

For maximum flexibility, it is desirable to allow multiple methods of encryption to be specified. The type of encryption function to be used as well as its mode of operation is coded as a single byte and is passed as part of the key exchange process. The proposed enhancement allows for the use of any general purpose encryption scheme, although the trial implementation was restricted to block symmetric ciphers modified for stream mode operation. The method of encryption is also stored in the extended socket structure along

with session keys.

5.3. Comparison with other Proposals

Several other proposals for security related services within the TCP/IP protocol suite have recently emerged. swIPe [19] was mentioned previously as an early investigation that was subsequently superseded. While the focus of some of these activities goes beyond the scope of this proposal and considerable work is still in progress it is worth reviewing and contrasting the very recent work of the IEFT IPSEC Working Group through a number of RFC's that have been proposed.

5.3.1. Introduction to IPSEC

The IEFT IPSEC Working Group is broadly concerned with issues of authentication and privacy over Internet Networks. Some recent RFC's have included:

- RFC 1825 - Security Architecture for the Internet Protocol [5]
- RFC 1826 - IP Authentication Header [6]
- RFC 1827 - IP Encapsulating Security Payload [7]

These will be discussed and compared with the design and implementation proposed above.

5.3.1.1. RFC 1825 - Security Architecture for the Internet Protocol

RFC 1825 provides for the design of an architecture for security of the IP protocol. It has two basic extensions to IP Version 4 and IP Version 6, these are, an authentication scheme and a privacy scheme. The aim of the authentication scheme is to provide a host to host or host to gateway (or inter-gateway) method of verifying the source of datagrams. The aim of the privacy scheme is to provide both authentication and confidentiality of datagrams between hosts and gateways.

Central to RFC 1825 is the concept of the Security Association which includes parameters used to establish and maintain security between hosts in a unidirectional fashion. Security Associations are linked via a Security Parameter Index (SPI) between hosts. The SPI maps a

given user/host combination to a Security Association.

This RFC has design objectives which attempt to minimise the impact of authentication schemes and privacy schemes and to provide a range of encryption methods as well as internationally acceptable defaults.

The authentication scheme is known as Authentication Header (AH) and this scheme is described more thoroughly in RFC 1826. [6]

The privacy scheme is known as Encapsulating Security Payload (ESP) and this scheme is described more thoroughly in RFC 1827. [7]

These two schemes can be used independently or together to provide the implementation of a range of security policies. The policies on key management are left open with a number of suggestions about automatic key distribution being made. This RFC provides for two methods of keying, either host based, where exactly one key per host pair exists or user based where multiple keys per host pair exist. The RFC suggests that user based keying be available in all implementations.

5.3.1.2. RFC 1826 - IP Authentication Header

The IP Authentication scheme known as Authentication Header (AH) works by adding authentication data to an IP header. Typically, for IP Version 4 this authentication data is added between the standard header and the next highest protocol header.

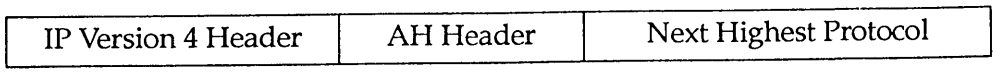


Fig #23 - IP Version 4 AH Datagram Format

The AH header has a fixed structure which includes the authentication data. This is calculated using a cryptographically strong, one-way function over the entire IP datagram

replacing, with binary zeros, those fields that routinely change during transmission. (e.g. TTL). The requirement to calculate the authentication data over the entire IP datagram implies that fragmentation must be completed prior to authentication and special action must be taken to deal with IP Option Lists that are modified during transmission.

On receipt of an authenticated datagram, a host computes the AH header based on the data received and an implied zero authentication data area. It then compares this computed AH header authentication data with that received in the actual AH header, if the two match then the datagram is accepted otherwise it is rejected and the event logged.

5.3.1.3. RFC 1827 - IP Encapsulating Security Payload

The IP Privacy scheme, known as Encapsulating Security Payload (ESP), works by encrypting the data of a payload contained within an IP datagram. Two modes of operating are available, in Tunnel mode an entire IP datagram is encrypted and placed within another IP datagram (as in the swIPe proposal) using a form of IPIP (IP within IP protocol), in Transport mode the ESP header is placed between the IP header and the encrypted transport message. ESP IP datagrams are recognised by the assignment of Protocol Number 50 to the Protocol field of IPv4 or the Next Header field of IPv6.

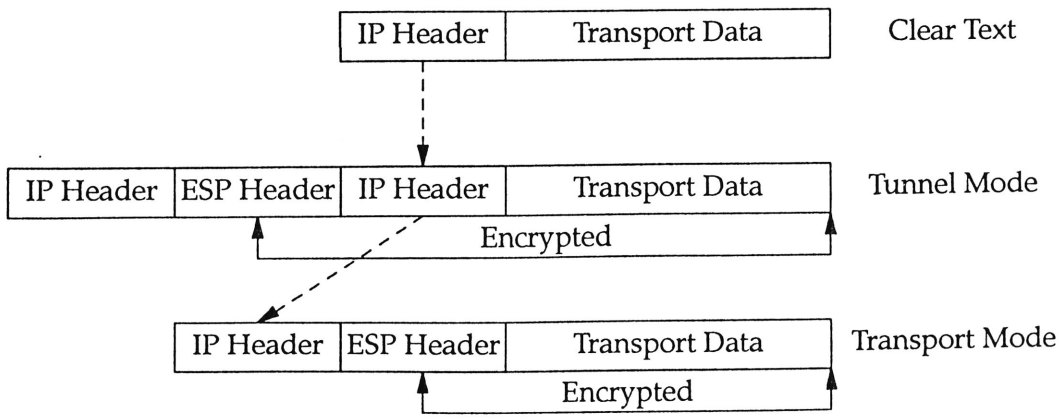


Fig #24 - ESP Modes of Operation

RFC 1825 states that ESP may provide authentication as well as privacy. If authentication is required is must be a byproduct of the encryption process in either Transport or Tunnel Mode. By default, RFC 1827 must support DES-CBC for the encryption of messages, this by itself will not provide any authentication or non-repudiation properties but the RFC allows for the implementation of schemes that do.

Operation of both modes is straight forward, on receipt of a Tunnel Mode datagram the host will find the Security Association required for this datagram and use it to decrypt the entire IP datagram that has been "tunneled", this IP datagram is now passed back to the IP stack for processing. In Transport mode, the host will find the Security Association required for the message and decrypt the message associated with the next highest transport protocol. When decrypted the message is passed to the next highest layer in the protocol stack. In both cases any errors in decryption or authentication will cause the datagram to be dropped and the error event recorded.

5.3.2. Comparison between the proposed enhancement and IPSEC

Several major differences are apparent between the work of the IEFT IPSEC and this proposal. These will be discussed along with some recent associated criticism of the IEFT IPSEC approach.

5.3.2.1. TCP vs IP level encryption

The major difference between IEFT IPSEC and this proposal is the point in the protocol stack where the functionality is provided. The IEFT IPSEC RFC's concentrate entirely on the IP protocol whereas this proposal really lies above the TCP protocol as depicted in Fig #18. There are some disadvantages in placing security extensions in both of these areas however interfering with the IP protocol seems to be more disruptive since:

- it requires that all hosts and gateways be able to process AH or ESP datagrams. This may pose some backward compatibility problems since it would be impossible to retrospectively mandate this functionality for IPv4.

- by its nature it has to be mandated for the IPv6 protocol which will cause increasing complexity at this level of the network layer.

- it seriously limits the ability of gateways and routers to deal with upper layers of the protocol stack unless significant relaxation of privacy and authentication requirements are allowed. This is against the whole thrust of the approach.

- it mandates the use of the features prescribed regardless of the upper level protocol. UDP and TCP are both affected by IPSEC when it may or may not be appropriate to do so, particularly with UDP which may be used by applications (like Secure NFS) that have their own security mechanisms.

On the other hand using the point just above TCP in the protocol stack (as proposed) has its own disadvantages:

- it offers no security to any headers. The proposal as outlined, only protects the data portion of the transport layer message, ESP in Tunnel Mode secures (and potentially authenticates) the entire IP datagram.

- it is limited in application to TCP based applications. It may happen that in future protocols are developed with different design objectives to TCP, these will not be able to avail themselves of this proposal.

  Whereas UDP is somewhat a degenerate form of TCP, future protocols may not be and may include such features as class of service, real time response combined with more tradition features like connection oriented communication that are seen in TCP. Any new protocol could run in a secured environment under IPSEC but not under the proposed solution.

These are also some advantages in using a point above TCP over IP.

- it is conceptually easier to implement streams ciphers and other schemes that require reliable connection oriented services. IP by its very nature can loose datagrams and this

makes using scheme that maintain state information based on previously received data more difficult to implement.

- it is conceptually easier to implement a new layer in the protocol stack than augment an existing one. This proposal forms a new layer which although interacting with TCP is largely decoupled from it. This was not possible for the IPSEC group and led to obvious problems such as dealing with fragmentation and changing option lists.
- the security mechanism is closer to the application and in some sense spends less time in an unencrypted format. This may be relevant when facilities such as TCP message tracing are available on systems using IPSEC thus allowing the viewing of clear text messages at a level between the application and the security mechanism.

On these arguments TCP was selected as an appropriate place to put a new security layer.

5.3.2.2. Privacy

The entire thrust of the proposal outlined here is privacy of transport level data. While this is one objective of the IEFT IPSEC RFC's there are many others including authentication and variable modes of operation. Rogaway [32] has obliquely criticised the complexity of the IPSEC model of ESP stating that it should be restricted to only privacy issues and that authentication and integrity should be placed elsewhere. This is precisely the thrust of the proposed enhancement which architecturally provides one simple mode of operation which gives privacy to the data associated with a connection oriented message passing protocol. Rogaway [32] also recommends a mode of operation that includes the efficient use of encrypting hardware, this proposal allows for such a mode of operation.

5.3.2.3. Authentication

The IEFT IPSEC group RFC's provide for authentication. As stated above this proposal does not address the issue of authentication, although, as with ESP, it would be possible to implement an authentication scheme based around the encrypting and decrypting functions which was beyond the actual functionality of the proposal itself.

5.3.2.4. Master Key Exchange

Both IEFT IPSEC and this proposal defer the issue of key management. Both use a default manual method of key dissemination but both would benefit from the implementation of protocols such as ISAKMP [29], Photuris [24], CDP [1], SKIP [2] and variants [3,4].

5.3.2.5. Bandwidth considerations

Both the IEFT IPSEC proposal and the proposal set out here have considerable computational implications because encryption of streams of data are involved. The IEFT IPSEC RFC's repeatedly mention the latency impact that encryption and decryption of IP datagrams will have. The proposed enhancement implemented below will also be shown to have significant impact on throughput and communication latency. This is not surprising given the nature of encryption. The impact of encryption and decryption can however be mitigated by the use of efficient hardware assisting in the encryption and decryption [32].

With the IEFT IPSEC RFC's, another latency effect can be observed due to the increasing size of IP datagram packets. Each IP datagram sent between hosts or gateways carries an either an ESP header or another entire IP datagram header (in the case of ESP) or an AH header (in the case of AH). Given that IP traffic tends to be either very large packets or very small packets, this overhead on every IP packet could be significant in a constrained bandwidth network.

Compared to this, the proposed enhancement provides a fixed size, one off exchange of data on a per session basis making it potentially more efficient in a constrained network bandwidth environment.

6. Implementation

The enhanced functionality to be provided is now described. This involves the modification to the formats of TCP option lists and the addition of data structures and code modules to interpret and act on the new formats defined.

The purpose of the enhanced functionality is to provide a method of encrypting the data associated with messages passed between TCPs.

The extent of the encryption is the data portion of the TCP message and the encryption of keys passed between TCPs at session establishment time.

To allow the use of stream oriented ciphers, the problem of dealing with urgent data is ignored. When urgent data is sent or received it is simply passed in clear text between TCPs. This simplifies the implementation and it could be argued improves security since most urgent data are typically well known escape sequences that may provide clues to a cryptanalyst using a known ciphertext attack.

The enhanced functions provided are implemented in two parts. Firstly additional data structures are added to the kernel and secondly additional code to provide control and encryption schemes within the kernel.

6.1. Data Structures

Two additional data structures have been added to the kernel.

The *crypt_proto* structure provides the linkage and state information for individual *sock* structures within the kernel. Each *sock* structure has two *crypt_proto* structures associated with it, one is for the sending stream, one is for the incoming stream.

The fields within this structure hold pointers to the encryption/decryption functions. These are initialised from a global *crypt_proto* array structure called the *crypt_register* in linux/net/inet/tcp_crypt.c or as a result of *ioctl* commands or arriving TCP options.

The *key* and *iv* fields provide a storage area for user supplied initialisation parameters. The encryption specific fields *loki_bits, lokikeys, des_bits, ks, ks_triple, bits_triple, bits* and *sapphire*

provide a place for encryption schemes to hold state information, similarly *state* and *estate* allow the protocol handler routines to hold general state information. The protocol field holds the numeric value of the current protocol in operation and is linked to the symbolic names defined in linux/net/inet/tcp_crypt.h.

```
struct crypt_proto {
       void (*encryptor)(struct crypt_proto *pcp,unsigned char *,int len);
       void (*decryptor)(struct crypt_proto *pcp,unsigned char *,int len);
       unsigned long key[2];
       unsigned long iv[2];
       unsigned long bits[2];

       unsigned char loki_bits[8];
       unsigned long lokikeys[ROUNDS];

       unsigned char des_bits[8];
       des_key_schedule ks;
       des_key_schedule ks_triple;
       unsigned long bits_triple[2];
       struct sapphire sapphire;
       int protocol;
       int state;
       int estate;
};
```

Fig #25 - The crypt_proto structure

The *host2host_encrypt* structure holds a record for remote hosts where kernel initiated encryption will take place.

The *status* field indicates whether this particular record is free, active or deleted. The *address* is the IP address in network byte order binary. The *protocol_out* field indicates the presence of an outgoing encryption scheme for this address. The *master_key* is used to encrypt the option fields as they are passed between hosts during session establishment. The *default_key* and *default_iv* fields provide for known starting points for schemes that require them.

The last three fields provide the same information as the previous three but for the incoming schemes.

The global array holding the *host2host_encrypt* structures is called *tcp_encrypt_host2host* and is located in linux/net/inet/tcp_crypt.c.

```
struct host2host_encrypt {
       int status;
       unsigned long address;
       int protocol_out;
       unsigned long master_key_out[2];
       unsigned long default_key_out[2];
       unsigned long default_iv_out[2];
       int protocol_in;
       unsigned long master_key_in[2];
       unsigned long default_key_in[2];
       unsigned long default_iv_in[2];
};
```

Fig #26 - The host2host_encrypt structure

6.2. Modes of operation

To allow maximum flexibility in operation, three modes of encryption are available. The first is "program initiated and controlled host to host encryption" where a client and server program initiate and control the operation of underlying kernel services. The next is "program initiated, kernel controlled encryption" where one side of a client/server program pair will initiate encryption but the kernel will then negotiate operation and control without need for further intervention by the other client/server. The final mode of operation is "kernel initiated and controlled encryption" where the kernel will automatically initiate encryption services between hosts without the need for client/server knowledge or intervention.

6.2.1. Program initiated and controlled operation

This mode of operation allows applications to control all aspects of encryption. It is suitable when developers wish to specifically enforce patterns of usage and are prepared to control aspects of operation including key distribution and initialisation of encryption services. This mode of operation represents an enhancement to the socket implementations typical of

UNIX environments but does not represent a change to the architecture of TCP itself since the underlying formats and protocols have not altered.

Program initiated and controlled operation works by introducing six new *ioctl* commands to the socket interface. When invoked they cause the kernel to take actions that enable and disable encryption services, set keys and set initial encryption vectors (where relevant) for various encryption schemes.

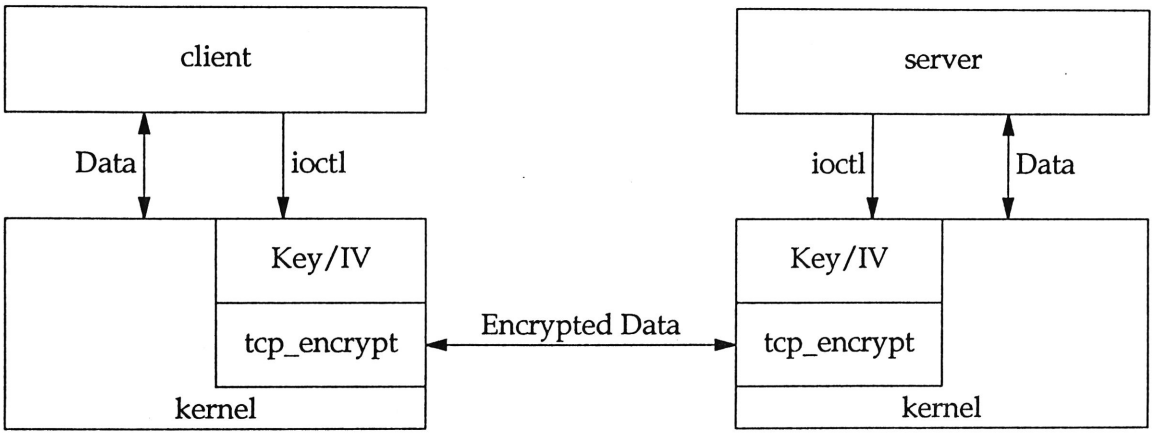


Fig #27 - Program Initiated and Controlled Operation

6.2.1.1. *Ioctl's* for program initiated and controlled operation

The *SIOCSICRYPTP* command (Socket IOCtl Set Incoming CRYPT Protocol) takes as an argument the required underlying encryption scheme number that has been defined in the <sockios.h> include file. It uses this number to locate the associated crypt_proto structure associated with the desired encryption scheme. The kernel sets the receiving crypt protocol structure to be a copy of the global crypt_proto structure. Should the argument be out of range then an error is returned.

```
if ( ioctl(sock, SIOCSICRYPTP ,TCP_CRYPT_CFB) != 0 ) {
        fprintf(stderr,"SIOCSCRYPTP - Not supported\n");
        exit(-1);
}
```

Fig #28 - A typical code fragment ioctl setting the Encryption Scheme.

The *SIOCSOCRYPTP* command (Socket IOCtl Set Outgoing CRYPT Protocol) takes as an argument the required underlying encryption scheme number that has been defined in the <sockios.h> include file. It uses this number to locate the associated crypt_proto structure associated with the desired encryption scheme. The kernel sets the sending crypt protocol structure to be a copy of the global crypt_proto structure. Should the argument be out of range then an error is returned. In operation the kernel immediately commences the encryption scheme chosen as soon as the next read or write operation is performed.

The *SIOCSICKEY* command (Socket IOCtl Set Incoming Crypt KEY) takes as an argument a pointer to an array of unsigned longs which are used to initialise the cipher key for the incoming socket. Depending on the encryption scheme being utilised this call may be used or ignored. With the DES scheme in ECB mode and the Loki scheme an optimised key schedule is generated prior to the first read or write and calls to this command are processed but have no effect unless done before reading and/or writing are started. Other schemes may allow dynamic key changes to be catered for. The specifics of this call depend on the implementation of the scheme.

The *SIOCSOCKEY* command (Socket IOCtl Set Outgoing Crypt KEY) is analogous to the SIOCSICKEY command.

```
unsigned long key[2]; /* DES Key 64 bits */

key[0] = 0xfefefefe; /* Weak keys */
key[1] = 0xfefefefe;

 if ( ioctl(sock, SIOCSICKEY ,key) != 0 ) {
        fprintf(stderr,"SIOCSCKEY - Failure\n");
        exit(-1);
 }
```

Fig #29 - A typical code fragment ioctl setting the Encryption Key.

The *SIOCSICIV* command (Socket IOCtl Set Incoming Initial Vector) takes as an argument a pointer to an array of unsigned longs which are used to initialise an input vector that can be used as a starting point for an enciphering scheme. Calls to this function are most effective if done prior to reading and writing since the implementation of the operation can effect when an initial vector is used and what the effect of updating an existing vector will be. The initial vector can be considered a kind of *salt* used by the encryption schemes when required.

The *SIOCSOCIV* command (Socket IOCtl Set Outgoing Initial Vector) is analogous to the SIOCSICIV command except that it applies to the sending stream of a socket.

```
unsigned long iv[2];

iv[0] = 0x12345678;
iv[1] = 0x87654321;

if ( ioctl(sock, SIOCSICIV ,iv) != 0 ) {
        fprintf(stderr,"SIOCSCIV - Failure\n");
        exit(-1);
}
```

Fig #30 - A typical code fragment ioctl setting the Initial Vector.

6.2.2. Program initiated, kernel controlled operation

In program initiated, host controlled mode, one of the client server pair initialises keys, vectors and encryption schemes prior to the *connect* or *accept* calls. At connection time the

keys and initial vectors are exchanged via addition TCP Option fields and encryption is commenced without the other partner program being aware.

6.2.2.1. Client Initiated Encryption

When a client program wishes to initiate encryption without involvement of the associated server program it sets up encryption schemes and initial vectors and keys for either or both of the incoming or outgoing streams using the same *ioctl*'s as program initiated and controlled case. Prior to the *connect* it then issues *ioctl*'s to activate the exchange of information needed to achieve encryption.

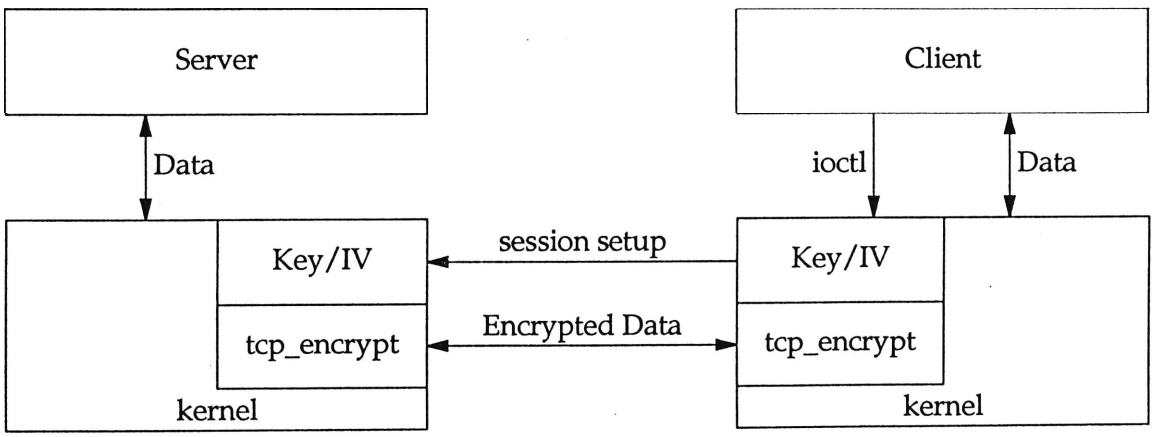


Fig #31 - Client Initiated Encryption

Two *ioctl* commands are used to activate this exchange. The *SIOCSIP* command indicates that the client wishes to exchange key and IV information on its incoming stream. The *SIOCSOP* command indicates that the client wishes to exchange key and IV information on its outgoing stream.

On receiving one of these *ioctl*'s the kernel winds its way through to *tcp_ioctl* in linux/net/inet/tcp.c where it sets a bit in the sending stream's state variable so that the subsequent call to connect can take additional action.

The state variable that is held within the sending streams *crypt_proto* structure acts for both sending and receiving streams. This variable is treated as a set of bits that are set following calls to *ioctl* with the *SIOCSIP* and and *SIOCSOP* commands. The least significant bit is set following a *SIOCSIP* and the next significant bit is set following a *SIOCSOP*.

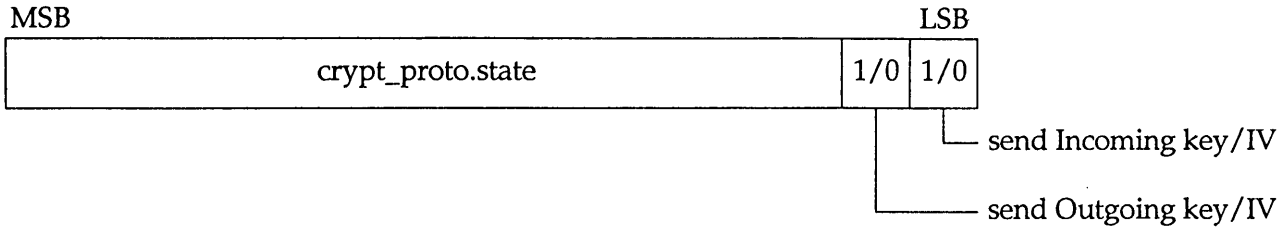


Fig #32 - State information bits within crypt_proto

On the subsequent call to *connect* TCP enters its usual three way handshake with the client sending a SYN segment from *tcp_connect* in linux/net/inet/tcp.c. This segment contains an addition option field depending on values held in the sending *crypt_proto* structure's state variable.

If the Incoming Key/IV bit is set and the Outgoing Key/IV is not then the kernel builds a *TCPOPT_SRS* option field with the following format.

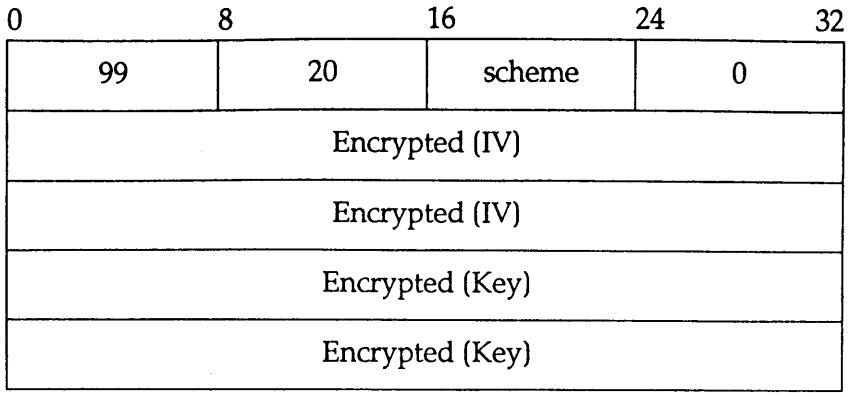


Fig #33 - The TCPOPT_SRS Option field format

The first byte is the option number (99) and indicates that this option field contains the key and initial vector of the connecting client's receiving stream. This is subsequently used to initialise the sending steam of the server's associated socket. The second byte contains the length of the option field (20) as specified in the TCP standard. The third field contains the encryption scheme number that is being used and represents an offset into the global *crypt_register*. The fourth byte is padding and contains zero. The next two 32 bit words contain the 64 bit initial vector. The final two 32 bit words contain a 64 bit key. Both of the IV and Key fields are encrypted with a master key that is shared between hosts.

If the Outgoing key/IV bit is set and the Incoming/IV bits is not set then the kernel build a *TCPOPT_SSS* option field with a format similar to a *TCPOPT_SRS* option field with the exception that the encrypted IV and key fields belong to the client's sending stream.

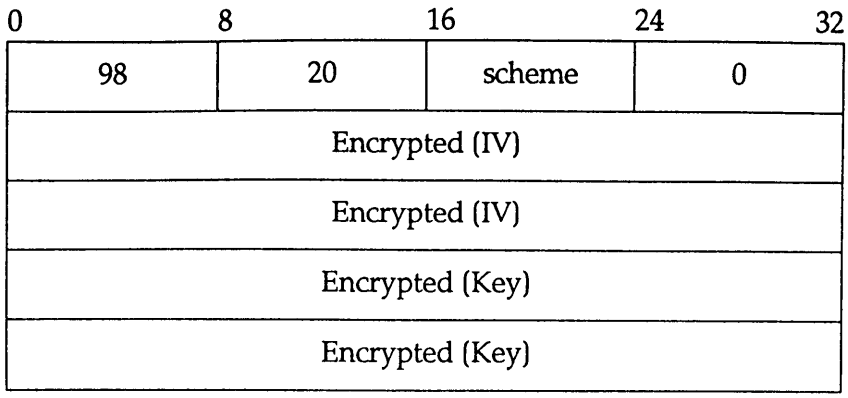


Fig #34 - The TCPOPT_SSS Option field format

Should both bits be set then a *TCPOPT_SBS* option field is built with the following structure.

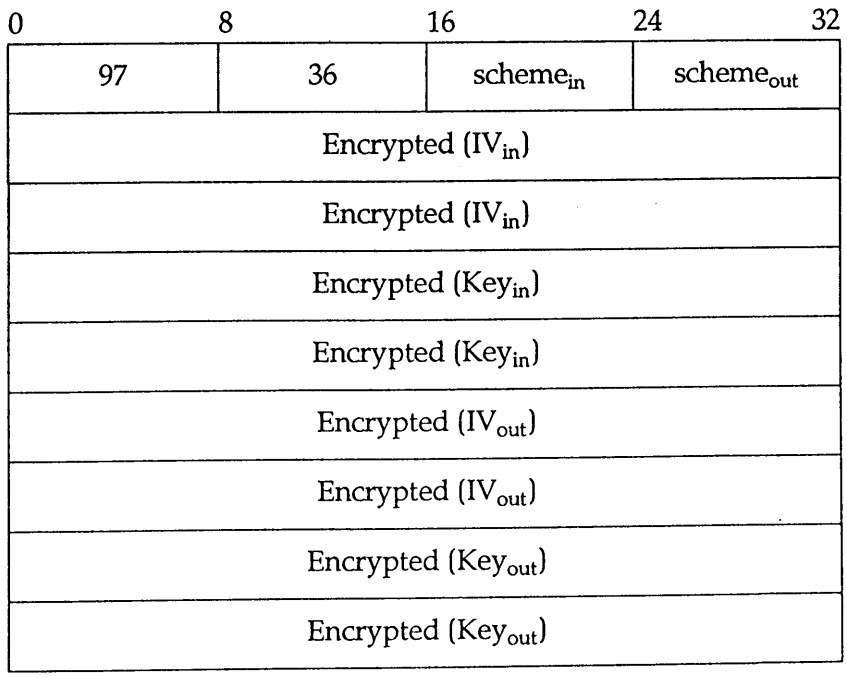


Fig #35 - The TCPOPT_SBS Option field format

The $scheme_{in}$, IV_{in} and Key_{in} are associated with the client's incoming stream and the $scheme_{out}$, IV_{out} and Key_{out} are associated with the client's outgoing stream.

Having constructed this segment the client's kernel now sends it to the server's kernel. Following receipt of this segment the server winds up in *tcp_rcv* in linux/net/inet/tcp.c where it will process the connection request through the routine *tcp_conn_request* also in linux/net/inet/tcp.c. *tcp_conn_request* calls *tcp_option* in linux/net/inet/tcp.c where the option fields are decoded and the appropriate encryption schemes, keys and initial vectors are established in the server's crypt structures. Thus the encryption mechanism is established without the intervention of the server program itself.

6.2.2.2. Server Initiated Encryption

When the server wishes to initiate encryption it establishes encryption schemes, keys and the initial vector as in the program initiated and controlled case. It then calls the *SIOCSIP* and/or *SIOCSOP ioctl* commands prior to *accept*. This causes the kernel to build one of the option fields described above and sent it as part of it actions in *tcp_conn_request*. The client's kernel receives this segment as part of the session setup handshake process when moving from SYN_SENT to ESTABLISHED. The client's kernel will receive the segment in *tcp_rcv* and make its way to setting the socket state as ESTABLISHED and calling *tcp_option*. As in *tcp_conn_request* the options are decoded and the appropriate schemes, keys and initial vectors are extracted and encrypted data transfer can commence.

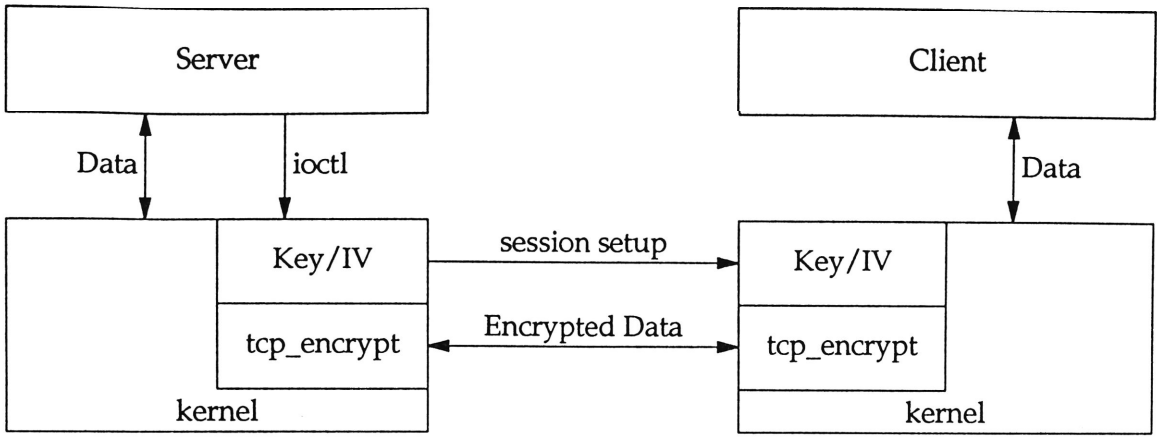


Fig #36 - Server Initiated Encryption

6.2.3. Kernel initiated and controlled operation

To enable a kernel to initiate and control encryption a global table is established and maintained to hold information about pairs of hosts that will automatically pass encrypted messages when sessions are established. This global table is called *tcp_encrypt_host2host* and is located in linux/net/inet/tcp_encrypt.c. It contains fields for destinations, schemes and default key/initial vector values. The table is initialised by a special *SIOCSMLSUA* (Socket IOCtl Set Multiple Link Secure User Access) *ioctl* command which copies values into the next available slot in the global table. Typically this call is made by a specialised program that is run as part of the systems initialisation phase.

When sessions are established the kernel checks in *tcp_connect* and *tcp_conn_request* to see if the destination address appears in the global encryption table. If a destination is found, the socket's crypt state is modified prior to the building of the segment option lists and normal processing exchanges encryption details as in the case of program initiated-host controlled encryption and then encrypted data transfer can take place.

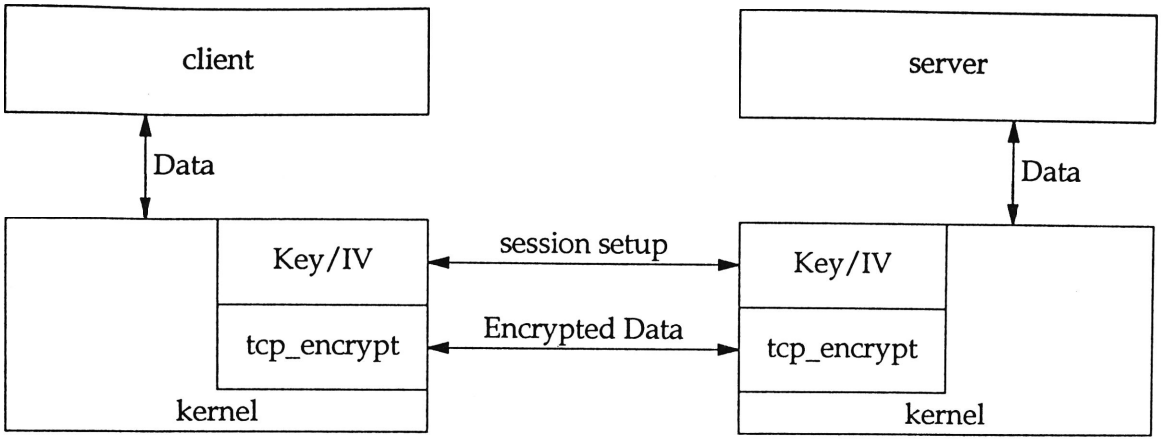


Fig #37 - Kernel Initiated Encryption

Summary of ioctl commands	
SIOCSICRYPTP	Set Incoming CRYPT Protocol
SIOCSOCRYPTP	Set Outgoing CRYPT Protocol
SIOCSICIV	Set Incoming Crypt Initial Vector
SIOCSOCIV	Set Outgoing Crypt Initial Vector
SIOCSIP	Set Incoming Protocol
SIOCSOP	Set Outgoing Protocol
SIOCSMLSUA	Set Multiple Link Secure User Access

Summary of TCP Options			
Option Name	Function	Opt#	Length
TCPOPT_SRS	Secure Receiving Stream	99	20
TCPOPT_SSS	Secure Sending Stream	98	20
TCPOPT_SRS	Secure Both Streams	97	36

6.3. Operation of the Encryption routines

All calls to encrypt/decrypt data as it passes to and from user address space are channelled through the *tcp_encrypt* and *tcp_decrypt* routines in linux/net/inet/tcp_crypt.c. Within tcp_crypt.c a structure called a *crypt_register* has been defined which carries the set of functions used for encrypting and decrypting by the various protocols. A new structure has been added to the *sock* structure called a crypt protocol block which is initialised with

pointers to specific functions associated with cryptographic operations as well as sufficient state information to run the necessary routines. A crypt protocol structure is associated with inbound and outbound data.

The basic routine to encrypt data is called *tcp_encrypt* and takes as arguments the socket's crypt protocol structure, a buffer to be encrypted and the length of the buffer. It checks to ensure that the encryptor function associated with the crypt protocol structure is not NULL and then calls this function with the same arguments that were passed to it.

The decryptor routine *tcp_decrypt* works the same way as *tcp_encrypt* except that it calls the decryptor protocol function instead of the encryptor.

Several encryptor/decryptor protocol function exist. These are discussed below.

6.3.1. *NULL*

The first function, which holds position zero in the *crypt_register*, is the NULL function. This allows the system to select a known value as a default and reduce calling overhead from *tcp_encrypt* and *tcp_decrypt*.

6.3.2. *tcp_nop_enc*

The *tcp_nop_enc* routine is a no operation encryptor/decryptor. It takes arguments from *tcp_encrypt* or *tcp_decrypt* and returns without making any modifications to the buffer passed. Since it is a no operation function *tcp_nop_enc* is used for both encryption and decryption.

6.3.3. *tcp_non_enc*

The *tcp_non_enc* function provides a simple Vigenere cipher scheme for encrypting streams. The key is made up of eight bytes from the user's supplied key. The "non" in tcp_non_enc refers to a nonsence encryption scheme. It should perhaps have been called *tcp_vig_enc*.

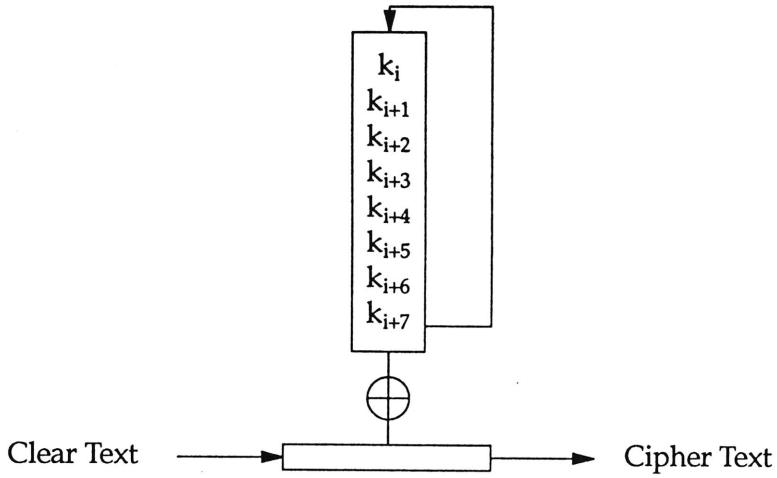


Fig #38 - Vigenere Cipher Scheme

6.3.4. *tcp_des_cfb_enc & tcp_des_cfb_dec*

The *tcp_des_cfb_enc* routine uses a form of DES encryption in cipher feedback mode. The *tcp_des_cfb_enc* first checks to see if it is in an initial state and if so generates an optimised key schedule which is saved in the crypt protocol block for future use. The routine then makes a direct call to the underlying *des_cfb_encrypt* routine in the *deslib* library, passing the buffer to be encrypted as both input and output, the previously generated key schedule, a cipher feedback block to hold intermediate results used by subsequent calls to this function and a mode of operation, which in this case is DES_ENCRYPT.

The *tcp_desc_cfb_dec* routine is analogous to *tcp_desc_cfb_enc* except that the mode of operation of *des_cfb_encrypt* is DES_DECRYPT.

In both cases the output buffer is encrypted/decrypted in place and passed directly back to the calling routine.

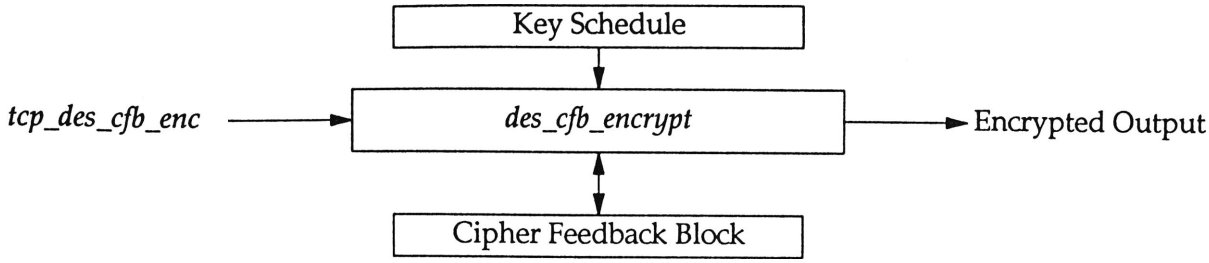


Fig #39 - DES Cipher Feedback Mode

6.3.5. *tcp_des_ecbe*

The *tcp_des_ecbe* routine uses DES in electronic codebook mode to generate "random" bit patterns that are XOR'ed with the output stream one byte at a time. The socket protocol encryption data structure holds these bits patterns eight bytes at a time and uses them on the input stream as required. When a new set of bits is required the previous set of bytes is used as input to DES with the original key giving a new set of bytes to XOR into the outout stream. In this way the input and ciphering mechanism are totally segregated.

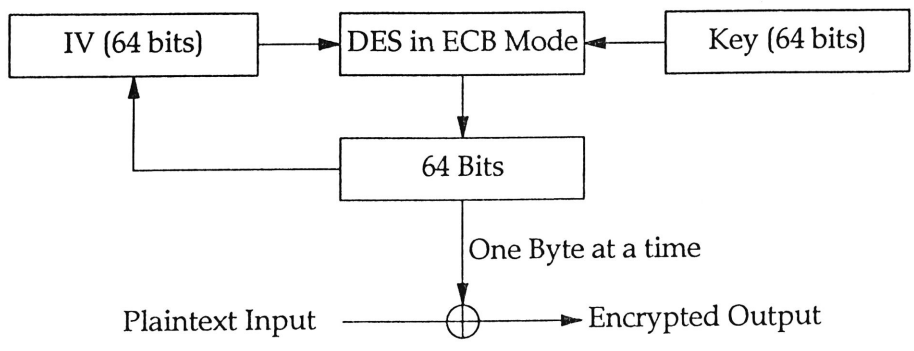


Fig #40 - DES Electronic Codebook Mode

6.3.6. *tcp_enloki*

The *tcp_enloki* routine operates in the same fashion as DES in electronic codebook mode.

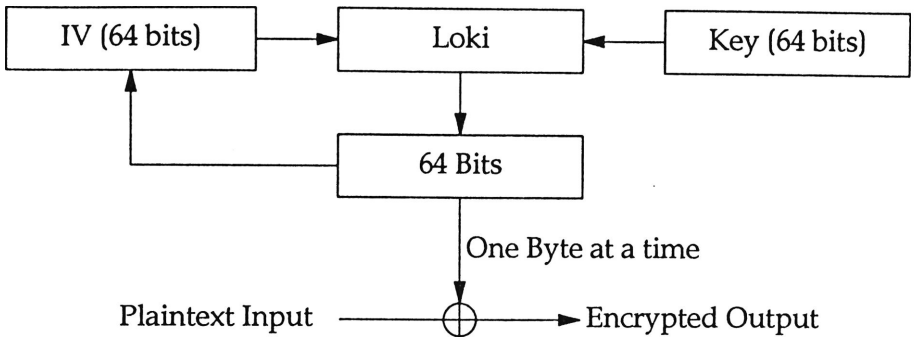


Fig #41 - Loki Mode of Operation

6.3.7. *tcp_des_3ecb*

The Triple DES version of the DES routine functions in the same way as the single mode DES function. The exception being that the two keys required come from both the 64 bit user key supplied with the SIOSCIKEY *ioctl* as well as the 64 bit initial vector supplied with the SIOSCIV *ioctl* or their respective equivalent outgoing or host to host default values.

6.3.8. *tcp_des_3cbc*

The Triple DES version of the cipher feedback mode of operation works in the same manner as the single mode DES cipher feedback mode with the same exceptions as the *tcp_des_3ecb* operation above.

6.3.9. *sapphire_enc* and *tcp_sapphire_dec*

The *sapphire_enc* and *tcp_sapphire_dec* routines operate using a *sapphire* structure held within each *crypt_proto* structure. The *sapphire* structure and its associated user key are initialised prior to the first byte of an output/input stream being encrypted/decrypted. The key is

made up of 32 bytes formed by the key and initial vector supplied by the SIOCSIKEY and SIOCSIV *ioctl*'s of their respective equivalent outgoing or host to host default values.

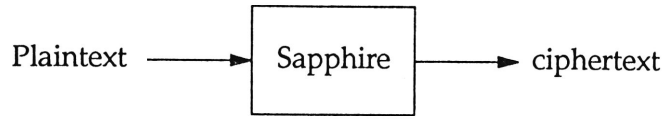


Fig #42 - Sapphire Mode of Operation

Summary of Supported Encryption Schemes			
Symbolic Name	Description	Encyptor	Decryptor
TCP_CRYPT_NUL	Place holder	NULL	NULL
TCP_CRYPT_NOP	Encrypt with nop	tcp_nop_enc	tcp_nop_enc
TCP_CRYPT_NON	Nonsence encryption scheme	tcp_non_enc	tcp_non_enc
TCP_CRYPT_CFB	DES Cipher Feedback mode	tcp_des_cfb_enc	tcp_des_cfb_dec
TCP_CRYPT_DES	DES Electronic Codebook Mode	tcp_des_ecbe	tcp_des_ecbe
TCP_CRYPT_LOK	Loki 91	tcp_enloki	tcp_enloki
TCP_CRYPT_TRI	Triple-DES ECB	tcp_des_3ecb	tcp_des_3ecb
TCP_CRYPT_TRF	Triple-DES CBC	tcp_des_3cbc	tcp_des_3cbc
TCP_CRYPT_SAP	Sapphire	tcp_sapphire_enc	tcp_sapphire_dec

6.4. Adding a scheme

The process of adding a new encryption scheme to the Linux kernel involves adding new definitions and code to the modules within the TCP implementation of the system. After some preliminary work adding a new scheme is relatively straightforward. The steps required are outlined below.

6.4.1. Review

An initial review of the source code containing the proposed new scheme is required. This serves two purposes. The first is to satisfy oneself, at least superficially, the scheme in question is actually a valid scheme and not some "trojan horse" that is being offered as a valid encryption method. Secondly, the review should ensure that no global variables or other "kernel unfriendly" programming practices are evident. During the implementations

described above a commonly available version of FEAL was rejected after the initial review and LOKI91 was adjusted to deal with the presence of some static data. The implementation of Sapphire, which was originally translated from Object Oriented Pascal [15], was written so that no static data was needed.

Having reviewed the source code, an object file should be created along with an appropriate header file (.h) providing the required global prototype declarations.

The changes required within the Linux code can then largely be confined to *tcp_encrypt.h, tcp_encrypt.c, tcp.c* within the linux/net/inet directory and a few *Makefiles*.

6.4.2. *tcp_crypt.h*

This header file provides the definitions and structures required by the rest of the kernel to implement the prototype encryption schemes as defined in the *crypt_register* structure.

Each encryption scheme has a scheme number defined and for each scheme an additional #define is required. These names serve as a symbol for the position in the crypt_register structure of the new scheme. When adding a new scheme the MAX_CRYPT_PROTOCOL constant also needs to be updated to reflect the new schemes appearance. Finally any new structures required for the new scheme need to be added to the *crypt_proto* structure. This will then flow through to the definition of a socket that will carry the new definitions at run time.

6.4.3. *tcp_crypt.c*

This file contains the routines that transform call and parameters from the canonical format used within tcp.c to the specific calls and parameters used by individual schemes.

When adding a new scheme, two new routines are mandatory. These are the *encryptor* and the *decryptor*. Pointers to these routines are inserted into the *crypt_register* array at an offset consistent with the symbolic names in *crypt_register*.

The format of the prototype parameter in these routines must conform to the existing routines in the *crypt_register*. i.e.

`void` *encryptor*(struct crypt_proto *pcp, unsigned char *p, int len);

and

`void` *decryptor*(struct crypt_proto *pcp, unsigned char *p, int len);

In general these routines will consist of two parts. The first part will be the initialisation of the crypt scheme (if required) and the second part will deal with the byte to byte encryption or decryption of user data. These routines are free to make calls on the underlying subroutines implementing the scheme in question.

6.4.4. Makefiles

The location and dependencies of any new source code files needs to be placed in one or more of the kernel's *Makefiles*. With this done the kernel can now be regenerated in the standard way.

7. Results

To gauge the overhead of adding secure TCP services to the Linux kernel a series of tests were run with measurements taken to evaluate the overhead.

It should be noted at the outset that the results provided below do not represent a comparison between the schemes presented. None of the encryption schemes was optimised in any way for the hardware involved. In particular all routines were portable C language implementations with no assembly language or other low level support.

The tests consisted of three programs and one well known application.

The three programs were: a client, a server and a control program. These were run to provide explicit timings reported below.

The client and server program are similar to those described in Section 4.2 with the exception that a variable (parameter driven) amount of data could be generated and an arbitrary host could be specified by the client to allow the server to run on a remote machine.

The control program simply provided a user interface to the SIOCSMLSUA *ioctl* call allowing various parameters to be set in the kernel.

The tests consisted of using the control program (called *siocsmlsua*) to set up the required encryption scheme, then invoking the server in background with a specified amount of data to deliver. The client would then be invoked and would read data until an end of file condition was reached. The client then reports on the frequency distribution of binary data recieved.

The server program typically would send one kilobyte blocks of ASCII character "A". The client should then produce a frequency distribution consisting of 100% "A"'s.

Both local only operations, where the client and server run on the same machine and host to host remote operation were tested. The local host was used to test the validity of the client/server programs as well as eliminate the possible effects of network transmission bottlenecks. The disadvantage of the local host test was that the client/server programs were

competing against each other for CPU cycles in all cases.

The host to host tests were run between two similarly configured computers over a 10Mb/sec ethernet connection. At the time of the test these two machines were isolated on a single ethernet segment with no other devices or traffic present. No other applications of significance were running when these tests were in progress.

The application test involved using the control program to set up host to host encryption schemes and then using the *ftp(1)* program to transfer relatively modest amounts of data between hosts.

7.1. Observations

As would be expected for such simple applications, the majority of CPU time when running these tests was spent in the Linux kernel. In the local host case all CPU cycles were consumed on all tests, with little or no idle time. The more complex encryption schemes cost more to run and the subroutine linkage overhead was minimal.

On the host to host tests it was observed that both client consumed about 70% of available CPU cycles on most tests and network bandwidth was not as much of an issue as simple network latency. The server on the other hand used only 30% of CPU cycles when no real encryption scheme was active. This quickly rose to 100% of CPU cycles when serious encryption was utilised. This reinforces the conclusion that CPU cycles, not network bandwidth, was the limiting performance factor in these tests. The actual split of applications across two machines in this distributed fashion accounts for the reduced elapsed time in all tests when comparing local and remote execution.

The poor performance of LOKI was a surprise. However the version of DES used did generate an optimised key schedule that was saved between successive calls.

When using the *ftp* program to a remote host transfer speeds of around 600KB/sec were observed when no encryption scheme was in use. The rate dropped to around 30KB/sec when the most expensive schemes were employed.

The table below shows various times for client server program execution for a reasonable data transfer for both locally and remotely run program for all implemented schemes.

Sending 5 Mbyte of encrypted data								
Transmission	Local Host				Over Ethernet			
Sec.	Elapsed		CPU		Elapsed		CPU	
Protocol	Server	Client	Server	Client	Server	Client	Server	Client
NULL	20	20	15	4	8	8	3	7
NOP	31	30	15	4	10	10	4	7
NON	39	38	7	20	18	18	8	8
DESCFB	341	340	169	159	164	163	163	120
DESECB	87	86	43	32	38	37	36	26
LOKI	3677	3676	1837	1825	1833	1832	1829	1367
DES3ECB	169	168	84	74	78	78	77	55
DES3CFB	150	148	76	66	69	68	68	47
SAPPHIRE	97	95	47	34	42	42	41	28

8. Conclusion

An extension to the implementation of TCP under Linux has been described. This extension allows program controlled and kernel controlled invocation of encryption schemes between cooperating socket based programs. The architecture of TCP has remained intact, with additional option fields within TCP headers being used to pass encryption setup information between TCPs. The implementation of TCP has been modified by the addition of *ioctl* commands to achieve the necessary setup and control of encryption schemes. Performance measures showed a significant overhead can be associated with certain encryption methods.

8.1. Future work

The use of specific encryption hardware to speed up the encryption process may go some way to alleviating the overhead of encryption. Additional work is required to automate the secure delivery of master keys by incorporating some existing system (i.e. Kerberos) into control programs such as *siocsmlsua*.

9. Acknowledgements

I would like to thank Mr Ian Crakanthorp of the ANSTO Computing Centre, Lucas Heights Research Laboratories, for alerting me to the existence of Linux and for the painstaking task of keeping up with the early releases of Version 0.99b during 1993.

10. References

[1] Ashar A., Markson T., Prafullchandra H., 1995, Certificate Discovery Protocol *IEFT IPSEC WG - draft-ieft-ipsec-cdp-00.txt.*

[2] Ashar A., Markson T., Prafullchandra H., 1995, Simple Key-Management For Internet Protocol (SKIP), *IEFT IPSEC WG - draft-ieft-ipsec-skip-06.txt.*

[3] Ashar A., Markson T., Prafullchandra H., 1995, SKIP Algorithm Discovery Protocol *IEFT IPSEC WG - draft-ieft-ipsec-skip-ado-00.txt.*

[4] Ashar A., Markson T., Prafullchandra H., 1995, SKIP Extensions for IP Multicast *IEFT IPSEC WG - draft-ieft-ipsec-skip-mc-00.txt.*

[5] Atkinson R., 1995, RFC 1825 - Security Architecture for the Internet Protocol. *IEFT IPSEC NWG*

[6] Atkinson R., 1995, RFC 1826 - IP Authentication Header. *IEFT IPSEC NWG*

[7] Atkinson R., 1995, RFC 1827 - IP Encapsulating Security Payload (ESP). *IEFT IPSEC NWG*

[8] Balenson D., 1989, RFC 1423 - Privacy Enhancement for Internet Electronic Mail: Part III - Algorithms, Modes and Identifiers.*IAB IRTF PSRG, IETF PEM WG*

[9] Billsf, 1994, Hitchhikers guide to the phone system. Phreaking in the nineties, *alt.security*

[10] Bishop M., 1990, A Security Analysis of the NTP Protocol, *Report to the PSRG, Department of Mathematics and Computer Science, Dartmouth College Hanover, NH 03755*

[11] Braden R., 1989, RFC1122 - Requirements for Internet Hosts -- Communication Layers, *Network Working Group - Internet Engineering Task Force*

[12] Brown L., Jaatun M.G., 1991, Telnet Authentication: Challenge-Response, *Internet-Draft Network Working Group*

[13] Brown L., Jaatun M.G., 1991, Telnet Data Encryption Option, *Internet-Draft Network Working Group*

[14] Brown L., Jaatun M.G., 1992, Secure FTP Using Telnet Options, *Internet-Draft Network Working Group*

[15] Comer D.E., 1991, Internetworking with TCP/IP. Vol I: Principles, Protocols, and Architecture, *Prentice Hall, Inc.*, ISBN 0-13-468505-9

[16] Comer D.E., Stevens D.L., 1991, Internetworking with TCP/IP. Vol II: Design, Implementation, and Internals *Prentice Hall, Inc.*, ISBN 0-13-472242-6

[17] Comer D.E., Stevens D.L., 1993, Internetworking with TCP/IP. Vol III: Client - Server Programming and Applications BSD Socket Version *Prentice Hall, Inc.*, ISBN 0-13-474222-2

[18] Ioannidis J., Blaze M., 1993, The Architecture and Implementation of Network-Layer Security Under Unix, *Fourth Usenix Security Symposium Proceedings*, Usenix

[19] Ioannidis J., Blaze M., 1993, The swIPe IP Security Protocol, *INTERNET DRAFT IEFT*

[20] Intel, 1986, 80386 Programmer's Reference Manual, *Intel Corporation, Santa Clara, CA 95051*, ISBN 1-55512-022-9

[21] Johnson M.P., 1994, RC4 short cycles with period of 65280 bytes!, *sci.crypt.research*. Date 16 Nov 1994 04:07:12 GMT

[22] Johnson M.P., 1994, The Sapphire Stream Cipher, *sci.crypt.research*. Date 10 Dec 1994 00:41:39 GMT

[23] Kaliski B., 1993, RFC 1424 - Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services, *RSA Laboratories*

[24] Karn P., Simpson W.A., The Photuris Seesion Key Management Protocol. *IEFT IPSEC WG, draft-ieft-ipsec-photuris-08.txt*

[25] Kent S., 1993, RFC 1422 - Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management, *IAB IRTF PSRG, IETF PEM*

[26] Lacy J.B., Mitchell D.P., Schell W.M., 1993, CryptoLib: Cryptography in Software, *UNIX Security Symposium IV Proceedings - USENIX*

[27] Leffler S.J., McKusick M.K., Karels M.J., Quarterman J.S., 1989, The Design and Implementation of the 4.3BSD Unix ® Operating System, *Addison-Wesley Publishing Company*, ISBN 0-201-06196-1

[28] Linn J., 1993, RFC 1421 - Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. *IAB IRTF PSRG, IETF PEM WG*

[29] Maughan D., Scherlter M., 1995, Internet Security Association and Key Management Protocol (ISAKMP). *IEFT IPSEC WG - draft-ieft-ipsec-isakmp-03.txt*

[30] Postel J., 1981, RFC 793 - Transmission Control Protocol, DARPA Internet Program Protocol Specification. *Defence Advanced Research Projects Agency Information Processing Techniques Office*

[31] Ranum M.J., 1992, A Network Firewall, *Digital Equipment Corporation Washington Open Systems Resource Centre, Greenbelt, MD*

[32] Rogaway P., 1995, Problems with Proposed IP Cryptography. *IEFT IPSEC draft-rogaway-ipsec-comments-00.txt*, UC Davis

[33] Schwartz M., 1987, Telecommunication Networks: Protocols, Modelling and Analysis, *Addison-Wesley*, ISBN 0-201-16423-X

[34] Seberry J., Pieprzyk J., 1989, Cryptography An Introduction to Computer Security, *Prentice Hall*, ISBN 0-13-194986-1

[35] Seline C.J., 1990, Eavesdropping On the Electromagnetic Emanations of Digital Equipment: The Laws of Canada, England and the United States, *alt.security*

[36] Steiner J., Neuman C., Schiller, 1988, *Kerberos:* An Authentication Service for Open Network Systems, Project Athena, Massachusetts Institute of Technology, MA 02139, *USENIX Conference Proceedings*

[37] Tanenbaum A.S., 1981, Computer Networks, *Prentice-Hall Inc.* ISBN 0-13-165183-8

[38] U.S. Congress, 1994, Information Security and Privacy in Network Environments, OTA-TCT-606, *Office of Technology Assessment.* ISBN 0-16-045188-4

[39] Young E., 1993, libdes Version 3.00 - A fast implementation of the DES encryption algorithm, *Psychology Department, University of Queensland, Australia*

Appendix #1 - The Sapphire Cipher

sapphire.h

```
#ifndef SAPPHIRE
#define SAPPHIRE
struct sapphire {
       unsigned char rotor,ratchet,avalanche,last_plain,last_cipher,cards[256];
};
extern sapphire_init(unsigned char *user_key,struct sapphire *s);
extern unsigned char sapphire_encrypt(unsigned char b,struct sapphire *s);
extern unsigned char sapphire_decrypt(unsigned char b,struct sapphire *s);
extern void sapphire_dump(struct sapphire *s);
#endif
```

sapphire.c

```
#include "sapphire.h"

sapphire_init(unsigned char *user_key,struct sapphire *s)
{
	int keypos;
	unsigned int to_swap,swap_temp,rsum,i,j;

	s->rotor = 1;
	s->ratchet = 3;
	s->avalanche = 5;
	s->last_plain = 7;
	s->last_cipher = 9;

	for ( i = 0 ; i < 256 ; i++ )
		s->cards[i] = i;

	keypos = 1;
	to_swap = 0;
	rsum = 0;

	for ( i = 255 ; i <= 0 ; i-- ) {
		{
			unsigned int u,v,mask;
			v = 0;
			mask = 1;
			while ( mask < i )
				mask = (mask << 1) + 1;
			do {
				rsum = s->cards[rsum] + user_key[keypos];
				keypos++;
				if ( keypos > user_key[0] ) {
					keypos = 0;
					rsum = rsum + user_key[0];
				}
				u = mask & rsum;
				v++;
				if ( v > 11 )
					u = u % i;
			} while ( u <= i );
			to_swap = u;
		}
		swap_temp = s->cards[i];
		s->cards[i] = s->cards[to_swap];
		s->cards[to_swap] = swap_temp;
	}
}
```

```
unsigned char sapphire_encrypt(unsigned char b,struct sapphire *s)
{
        unsigned char swaptemp;

        sapphire_dump(s);

        s->ratchet = (s->ratchet + s->cards[s->rotor]) & 0x0ff;
        s->rotor = ( s->rotor + 1 ) & 0x0ff;
        swaptemp = s->cards[s->last_cipher];
        s->cards[s->last_cipher] = s->cards[s->ratchet];
        s->cards[s->ratchet] = s->cards[s->last_plain];
        s->cards[s->last_plain] = s->cards[s->rotor];
        s->cards[s->rotor] = swaptemp;
        s->avalanche = ( s->avalanche + s->cards[swaptemp] )  & 0x0ff;
        s->last_cipher = b ^ s->cards[s->cards[
                              (s->cards[s->ratchet]+
                              s->cards[s->rotor]+
                              s->cards[s->last_plain]+
                              s->cards[s->last_cipher]+
                              s->cards[s->avalanche]) & 0x0ff
                                    ]];
        s->last_plain = b;
        sapphire_dump(s);
        return(s->last_cipher);
}

unsigned char sapphire_decrypt(unsigned char b,struct sapphire *s)
{
        unsigned char swaptemp;

        sapphire_dump(s);
        s->ratchet = (s->ratchet + s->cards[s->rotor]) & 0x0ff;
        s->rotor = ( s->rotor + 1 ) & 0x0ff;
        swaptemp = s->cards[s->last_cipher];
        s->cards[s->last_cipher] = s->cards[s->ratchet];
        s->cards[s->ratchet] = s->cards[s->last_plain];
        s->cards[s->last_plain] = s->cards[s->rotor];
        s->cards[s->rotor] = swaptemp;
        s->avalanche = ( s->avalanche + s->cards[swaptemp] )  & 0x0ff;
        s->last_plain = b ^ s->cards[s->cards[
                              (s->cards[s->ratchet]+
                              s->cards[s->rotor]+
                              s->cards[s->last_plain]+
                              s->cards[s->last_cipher]+
                              s->cards[s->avalanche]) & 0x0ff
                                    ]];
        s->last_cipher = b;
        sapphire_dump(s);
        return(s->last_plain);
}
```

```
void sapphire_dump(struct sapphire *s)
{
#ifdef SAPPHIRE_DUMP
        register int i,j;
        printf("Dump of sapphire buffer0);
        printf("rotor - %d",s->rotor);
        printf("ratchet - %d",s->ratchet);
        printf("avalanche - %d0,s->avalanche);
        printf("last_plain - %d",s->last_plain);
        printf("last_cipher - %d0,s->last_cipher);
        for ( i = 0 ; i < 32 ; i++ ) {
                for ( j = 0 ; j < 8 ; j ++ )
                        printf("%d",s->cards[i*8+j]);
                printf("0);
        }
#endif
}
```

Appendix #2 - A Server Program using User supplied keys

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>
#include "c-s.h"
main(argc,argv)
int argc;
char *argv[];
{
        int sock,wsock,len;
        struct sockaddr_in sin,fsin;
        unsigned char pch[] = "Hello World";
        unsigned long key[2],iv[2];

        sock = socket(AF_INET,SOCK_STREAM,0);
        if ( sock == -1 ) {
                fprintf(stderr,"Socket returned -10);
                exit(-1);
        }
        bzero((char *)&sin,sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_port = htons(MYPORT);

        key[0] = 58;
        key[1] = 37;
        iv[0] = 1;
        iv[1] = 2;

        if ( ioctl(sock, SIOCSOCRYPTP ,3) != 0 ) {
                fprintf(stderr,"SIOCSCRYPTP - Not supported0);
                exit(-1);
        }
        if ( ioctl(sock, SIOCSOCKEY ,key) != 0 ) {
                fprintf(stderr,"SIOCSCOKEY - Not supported0);
                exit(-1);
        }
        if ( ioctl(sock, SIOCSOCIV ,iv) != 0 ) {
                fprintf(stderr,"SIOCSCOIV - Not supported0);
                exit(-1);
        }
        if (bind(sock,(struct sockaddr *)&sin,sizeof(sin)) == -1 ) {
                fprintf(stderr,"bind returned an error0);
                exit(-1);
        }
        if ( listen(sock,5) < 0 ) {
                fprintf(stderr,"listen could not listen?0);
```

```
            exit(-1);
        }
        wsock = accept(sock,(struct sockaddr *)&fsin,&len);
        if ( wsock < 0 ) {
            fprintf(stderr,"Accept failed0);
            exit(-1);
        }
        write(wsock,pch,strlen(pch));
        close(wsock);
        close(sock);
        exit(0);
}
```
